Self-Adaptive Edge Services: Enhancing Reliability, Efficiency, and Adaptiveness under Unreliable, Scarce, and Dissimilar Resources

Zheng Song

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science and Applications

May 12, 2020

Blacksburg, Virginia

Keywords: Edge Computing, System and Programming Support, Execution Equivalence

Self-Adaptive Edge Services: Enhancing Reliability, Efficiency, and Adaptiveness under Unreliable, Scarce, and Dissimilar Resources

Zheng Song

(ABSTRACT)

As compared to traditional cloud computing, edge computing provides computational, sensor, and storage resources co-located with client requests, thereby reducing network transmission and providing context-awareness. While server farms can allocate cloud computing resources on demand at runtime, edge-based heterogeneous devices, ranging from stationary servers to mobile, IoT, and energy harvesting devices are not nearly as reliable and abundant. As a result, edge application developers face the following obstacles: 1) heterogeneous devices provide hard-to-access resources, due to dissimilar capabilities, operating systems, execution platforms, and communication interfaces; 2) unreliable resources cause high failure rates, due to device mobility, low energy status, and other environmental factors; 3) resource scarcity hinders the performance; 4) the dissimilar and dynamic resources across edge environments make QoS impossible to guarantee.

Edge environments are characterized by the prevalence of equivalent functionalities, which satisfy the same application requirements by different means. The thesis of this research is that equivalent functionalities can be exploited to improve the reliability, efficiency, and adaptiveness of edge-based services. To prove this thesis, this dissertation comprises three key interrelated research thrusts: 1) create a system architecture and programming support for providing edge services that run on heterogeneous and ever changing edge devices; 2) introduce programming abstractions for executing equivalent functionalities; 3) apply equivalent functionalities to improve the reliability, efficiency, and adaptiveness of edge services.

We demonstrate how the connected devices with unreliable, dynamic, and scarce resources can automatically form a reliable, adaptive, and efficient execution environment for sensing, computing, and other non-trivial tasks.

This dissertation is based on 5 conference papers, presented at ICDCS'20, ICWS'19, EDGE'19, CLOUD'18, and MobileSoft'18.

Self-Adaptive Edge Services: Enhancing Reliability, Efficiency, and Adaptiveness under Unreliable, Scarce, and Dissimilar Resources

Zheng Song

(GENERAL AUDIENCE ABSTRACT)

As mobile and IoT devices are generating ever-increasing volumes of sensor data, it has become impossible to transfer this data to remote cloud-based servers for processing. As an alternative, edge computing coordinates nearby computing resources that can be used for local processing. However, while cloud computing resources are abundant and reliable, edge computing ones are scarce and unreliable. This dissertation research introduces novel execution strategies that make it possible to provide reliable, efficient, and flexible edge-based computing services in dissimilar edge environments.

Dedication

To my family:

My Wife, Tianzi Wang

My Father, Laihui Song

My Mother, Yuhua Huang

Acknowledgments

I would like to sincerely thank all those who have helped me complete my Ph.D.

First, I would like to express my deepest appreciation to my advisor, Dr. Eli Tilevich, for his support and guidance throughout my doctoral studies. I am deeply impressed by his penchant for innovative thinking and brainstorming ideas with his students, as well as his tireless workaholism and strong self-discipline. His valuable and selfless advice not only makes me a better researcher, but also a better person, both mentally and physically. I am very fortunate to have an excellent mentor like him and I would like to thank him for an enriching Ph.D. experience.

I would also like to thank my committee members, Drs. Ali Butt, Christine Julien, Dimitrios Nikolopoulos, and Francisco Servant for their constructive feedback and valuable insights that made it possible for me to improve the overall quality of this dissertation.

My academic job search would not have been as successful if not for the recommendation letters and fruitful advice from Drs. Eli Tilevich, Shiyi Wei, and Christine Julien. It is thanks to their invaluable guidance and strong support, I am looking forward to commencing my own academic career this fall.

I am grateful to all CS@VT friends and Software Innovations Lab mates for their friendship and countless help. I was able to develop many research ideas and improve my communication skills by following their comments raised during seminars and technical discussions.

My graduate journey would not have been possible without my family. I would like to thank my family for their love, support, and encouragement. I would like to thank my wife, Tianzi Wang, for everything. Her encouragement and support have helped me overcome many obstacles in the past 6 years.

This research received support from National Science Foundation through the Grant 1717065 (SHF:CSR:Small:“Perpetuum Mobile: Orchestrating the Provisioning of Pervasive Resources for Emerging Mobile Applications”) and Grant 1649583 (SHF: EAGER: “Addressing Resource Scarcity via Distributed Mobile Services”).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The rapid growth of the Internet of Things (IoT) has changed multiple application domains, including healthcare, home, environment, and transportation. The increasing deployment of IoT sensors generates huge amounts of context-related sensor data to be processed by algorithms for sense-making, event monitoring, assisting humans, and automated control. In 2018, 17 billion of mobile and IoT devices were in use, producing more than 200 ZB of sensor data. By 2021, the connected things are predicted to rise to 25 billion, producing more than 800 ZB of sensor data [105, 121]. However, the estimated IP traffic will only be able to reach 20 ZB by that time. Even assuming that 10% of all generated sensor data is useful (i.e., 80 ZB), this amount still exceeds the cloud data center traffic by a factor of four [121]. Edge computing is expected to bridge this gap.

Edge computing processes massive amounts of sensing and personal data collected by mobile and IoT devices “at the edge of the network,” thereby reducing network transmission and communication latency. For example, a large scale video surveillance system can preprocess the videos and only upload the useful clips to the cloud, thereby reducing the network transmission load [155]. Augmented reality can boost its responsiveness by offloading the computationally intensive procedures to edge servers [134]. Moreover, edge computing devices often cooperate with IoT devices, to provide low-latency sense-making and smart control for manufacture, smart vehicle, and smart home environments [109].

Some edge systems are designed to provide domain-specific applications. For example, OPENVDAP [191] analyzes the driving data of autonomous vehicles; LAVEA [188] and VideoEdge [65] delegate video analytic tasks for nearby users; MUVR [94] aggregates the power of nearby devices used for VR gaming to accelerate their collective performance. Many other edge platforms provide system and programming support for edge applications and their developers, so an arbitrary edge application can be deployed and executed on demand in ubiquitous edge environments. Examples of such systems include Cloudlet [66, 95], PCloud [67], ParaDrop [98], and Cloud-sea [182].

These general-purpose edge platforms can be divided into two major categories, based on how they provide their resources. In the first category (e.g., Cloudlet and ParaDrop), edge resources are pre-deployed and sometimes managed by Internet Service Providers ([157]). In the second category (PCloud and Cloud-sea), edge devices at the scene (i.e., mobile and IoT devices, edge servers) form a collaborative computing environment [107, 123, 148] without involving a cloud-based controller. For these available edge devices to provide additional computational, networking, and sensor power, we must address the following challenges:

1. Existing system architectures lack safe and convenient support for mobile application developers to leverage opportunistic edge resources. While cloud computing has embraced the service-oriented architecture, the vast dissimilarities between cloud and edge environments make this architecture not directly applicable to edge-based solutions. For a mobile device to leverage the resources provided by its collocated edge devices, it needs to first allocate the appropriate devices energy- and latency-efficiently. Edge-based efficient resource allocation is particularly hard, as heterogeneous edge devices feature dissimilar network interfaces, application programming models, and system architectures. Besides, the executable code must either be pre-installed on the edge devices or be transferred from the requester device, thus reducing the utility or safety of edge computing.

2. Edge computing applications use the computational, sensor, and networking resources of nearby mobile and stationary computing devices. Because dissimilar devices can provide these resources, one cannot predict which exact combinations of resources will be available at runtime. The resulting variability renders edge applications unreliable and inefficient.

3. Fortunately, edge environments are characterized by ever-growing equivalent functionalities that satisfy the same requirements by different means. We explore the combined execution of equivalent functionalities to improve QoS. However, the existing service-oriented programming models cannot effectively orchestrate equivalent functionalities to execute them efficiently.

4. The vast dissimilarities in resource availability and capability across edge environments cause edge applications to suffer from inconsistent QoS.

How can one improve *the reliability, efficiency, and adaptiveness* of edge execution? This research's answer to this question is to exploit functional equivalence. Different implementations can satisfy the same functional requirement. For example, by implementing any of the following widely adopted methods, developers can satisfy the requirement of obtaining the user's geolocation [176]: 1) GPS-based: read the GPS sensor of the user's mobile device; 2) Cell-ID-based: read the unique Cell ID of a nearby cellular network access point, and retrieve the access point's location; 3) dead-reckoning-based: estimate the location by adjusting the last-obtained GPS location, as based on the user's motion, reported by the mobile device's motion sensors; 4) WiFi-fingerprint-based: collect the received signal strength (RSS) of the surrounding WiFi access points to form a fingerprint; compare it with a set of pre-trained location-marked fingerprints, and output the location of the closest fingerprint; and 5) some ad-hoc methods (e.g., camera-based, geomagnetic sensor-based, RFID-based localization, etc.). All these implementation options are *equivalent* in satisfying the requirement of obtaining the user's geolocation, albeit with vastly dissimilar non-functional characteristics, such as energy consumption, accuracy, reliability, and response latency.

1.1 Major Research Contributions and Scope

The thesis of this research is that functional equivalence can be exploited to improve the reliability, efficiency, and adaptiveness of edge-based services. Hence, our goal is to provide system and programming support for: 1) encapsulating the execution on edge devices as edge services; 2) leveraging equivalent functionalities in edge services to enhance reliability, efficiency, and adaptiveness. To achieve these goals, this dissertation includes three key interrelated research thrusts:

1. **To design and develop support for edge-service provisioning with heterogeneous and ever-changing edge devices**
   Edge computing leverages the resource capabilities (computation, storage, sensing, network, etc.) of co-located mobile devices at the edge of the network. However, enabling edge executions on heterogeneous mobile devices presents a number of difficulties, including the need to coordinate and steer the execution on devices with dissimilar resource capabilities, application programming models, and system architectures. Besides, the environment changes constantly due to the mobility and usage/energy status of co-located mobile devices. Hence, edge services provisioning with such heterogeneous and ever-changing edge devices requires system and programming support for discovering edge devices and finding a suitable device to execute a given functionality, as well as delivering the functionality's executable code to the discovered devices. We achieve this goal by comprehensively comparing an ad-hoc device-to-device approach and an

edge-gateway-based approach. These findings enable us to complete the basic design of an edge-based service provisioning system, to be used in the rest of this dissertation research [151, 154].

2. **To systematically support equivalent executions**

   Several software domains have applied equivalent execution to enhance various non-functional characteristics, including reliability, latency, cost efficiency, and accuracy [76, 82, 125, 184]. However, functional equivalence has never been fully supported as explicit programming abstractions that can be expressed, compiled, and executed. Although some languages (e.g., BPEL [103], Orc [77]) provide built-in support for predefined execution strategies, the increasing number of functionalities in each equivalent set can make it impossible to maximize performance via predefined execution strategies. To systematically support equivalent executions, this dissertation explores 1) a dedicated programming abstraction for declaratively specifying flexible execution strategies for equivalent functionalities; 2) how execution strategies can be generated automatically to fulfill given non-functional requirements; 3) new equivalence workflow constructs to execute equivalent functionalities efficiently [152].

3. **To apply equivalence for enhancing reliability, efficiency, and adaptiveness**

   Some edge resources are supplied by mobile or energy harvesting devices, which are failure-prone because of their mobility and limited energy budgets. To improve reliability and efficiency, we first explore how to enable edge service developers to specify equivalent functionalities and their execution strategies [153]. The vast resource dissimilarity across edge execution environments causes these executions to yield different QoS, so a service that follows the same execution strategy across pervasive environments is bound to deliver unpredictable and inconsistent QoS to the user. To improve adaptiveness, we involve a feed-back loop in the edge system design, to collect the environment-specific non-functional attributes of equivalent functionalities and generate execution strategies for given QoS requirements.

1.2 Broader Impact

This dissertation research is concerned with providing system and programming support for mobile application developers, with the goal of enabling them to efficiently leverage distributed resources at the edge of the network. The resource provisioning of edge systems is fundamentally different from that of cloud systems, as edge systems rely on the available resources provided by a variety of stakeholders at runtime, instead of allocating resources statically pre-deployed by vendors. The heterogeneity, resource scarcity, and unreliability of edge devices make the performance tuning methodologies for cloud-based distributed systems no longer applicable to edge-based systems. Meanwhile, the availability of equivalent functionalities provides unique but currently not utilized opportunities for enhancing the performance of edge applications. By tapping into the full potential of edge computing, ubiquitous IoT and mobile devices can collect enormous volumes of environmental and user data, to be processed by ever more sophisticated algorithms to enable novel intelligent applications. Besides, the findings presented in this dissertation can further be explored in other opportunistic computing environments with dynamic, scarce, and heterogeneous resources.

1.3 Structure

The rest of this dissertation is structured as follows. Chapter 2 compares multiple edge platforms and their programming supports. Chapter 3, 4, 5, and 6 covers our system design for edge-based service provisioning, programming support for enhancing the reliability, efficiency, adaptiveness of edge services, as well as programming support for equivalent functionalities. Chapter 7 presents concluding remarks, summarizes the contributions of this research, and discusses future work directions

Chapter 2

Literature Review

In contrast to cloud computing, edge computing utilizes local resources, deployed at the edge of the network. With these local resources offering low-latency low-cost near-field wireless communication, edge computing is also considered a key enabler of the emerging 5G standard [61, 157]. An important variant of this computing modality is *mobile edge computing* [54, 95] (also known as *collaborative edge computing* [148]), in which nearby mobile devices perform distributed computational tasks. In this chapter, we first summarize main application scenarios of edge computing, and then compare and contrast the state-of-the-art approaches that provide runtime and programming support for edge computing.

2.1 Edge Computing Application Scenarios

Edge computing has been applied to fulfill the requirements in several emerging application scenarios that can be roughly categorized as follows: latency-sensitive, data-intensive, and privacy-sensitive. Some scenarios fall into more than one of these categories. We describe these categories in turn next.

2.1.1 Latency-Sensitive Applications

A typical example of this category of edge applications is VR/AR gaming. The human response time in mobile games can tolerate a latency of at most 100 milliseconds, while immersive VR and AR can only tolerate around 20 milliseconds [35]. Li et al. reported that the average round-trip time from 260 global vantage points to their optimal Amazon Elastic Compute Cloud (EC2) instances is 74 ms, while Ha et al. demonstrated that the average round-trip latency of transmitting to a nearby edge server is twice as fast as transmitting to Amazon EC2. Based on these observations, edge computing has been explored in VR/AR gaming [189]. Other examples of such latency-sensitive applications include edge-enabled self-driving cars [97, 99] and industrial IoT[146].

2.1.2 Data-Intensive Applications

With the emergence of the Internet of Things (IoT), more and more data collection devices are generating vast amounts of local context-specific data. Transmitting all this data to cloud servers for analysis has become infeasible, so the gap between the volumes of locally generated data and the WAN throughput gave rise to another important category of edge-computing applications — data-intensive applications. For example, to avoid transmitting raw high-quality video streams to cloud servers, Long et al. studied how to cooperatively process video for IoT systems in edge networks; Tortonesi et al. designed a service model for processing IoT data in edge/fog networks; Team is an open-source framework, hosted by the Linux Foundation, that provides a common open platform for IoT edge computing. EdgeX involves an edge gateway that coordinates nearby data collecting devices, and deploys data processing methods at nearby computing devices.

To more efficiently process the ever increasing volumes of sensing data, a recent development of machine learning— federated learning [79]—enables deep learning models to be updated in a decentralized manner. Instead of requiring all raw sensing data to be transmitted to a cloud-based server, federated learning trains a model across multiple decentralized edge devices that have collected the sensing data, so these devices never share their raw data with each other.

2.1.3 Privacy-Sensitive Applications

Another increasingly common category are applications in which users place a high demand on preserving their privacy. In smart home environments, ubiquitous sensors collect personal data (e.g., user activities, preferences, and whereabouts), to be analyzed to provide better services to the user. Some users may not feel comfortable to upload their private data via a public network to a third-party remote server. Hence, some smart home applications utilize edge computing to prevent private data from being transmitted out of the household [162, 164]. Another typical usage scenario is mobile health, with its emphasis on protecting patients privacy [25].

2.2 Edge Computing System Designs

Edge computing systems have been designed in a variety of ways. These designs target dissimilar usage scenarios, relying on the presence of different system components. Next, we describe the main designs on the examples of several well-known edge systems, including: 1) Cloudlet [66]; 2) CloudPath [117]; 3) PCloud [67]; 4) Paradrop [98]; 5) SpanEdge [139]; 6) Cloud-Sea [182]; 7) Cachier and Precog [38]; 8) FocusStack [5]; 9) AirBox [14]; 10) Firework [192]. To describes these designs, we categorize them based on their handling of the following design questions.

2.2.1 What functionalities does an edge system provide?

Most existing edge systems are designed for storing and processing sensing data. For example, Cloudlet demonstrates its applications in cognitive assistance systems, IoT data analysis, and hostile environments. CloudPath is also designed for storing data along the path from edge to cloud, while providing data aggregation, caching, and processing services. PCloud stores and processes personal data at the edge. Precog provides caching services for recognition applications.

Some other edge systems further integrate sensing and sensor control capabilities. ParaDrop and Cloud-Sea are designed with considerations for connecting the pervasive resources in IoT, smart home, and industrial Internet of Everything environments. Some edge systems further provide networking capabilities. Examples include SpanEdge, in which the network capability is considered in scheduling the streaming of raw sensing data.

2.2.2 Whom an edge system benefits?

Some edge systems are designed to enhance the execution of nearby mobile devices, while others are designed to enhance cloud-based applications. The first category includes Cloudlet, PCloud (both accelerate mobile applications), and Precog (accelerates mobile recognition tasks). The second category includes CloudPath, ParaDrop, SpanEdge, and FocusStack. These two categories mainly differ in whether they rely on a central server to coordinate their execution. The systems in the second category all feature a central server (or say central controller), while those in the first category are designed for decentralized coordination. Cloud-sea combines these two categories: it provides both edge-based functionalities for mobile applications and data analysis functionalities for cloud applications.

2.2.3 Where does processing take place?

Based on the answer to this question, we categorize edge systems into four types:

- *Entirely at local/edge devices:* Examples include Cloudlet, Precog, and FocusStack. Cloudlet uses dedicated edge servers, one hop away from mobile devices. This design ensures that the communication latency between Cloudlet and its clients is low and that they share the same context. Precog uses edge servers to request prefetching, and FocusStack deploys containers on edge devices.

- *At both edge and cloud:* Examples include PCloud and ParaDrop. Such systems divide the tasks among the available edge devices and the cloud to maximize QoS characteristics, including latency, privacy, and throughput. An interesting property of these systems is their fairly straightforward system architecture, due to their uniform treatment of both edge and cloud computing resources.

- *At edge, cloud, and in-between:* Examples include CloudPath and SpanEdge. CloudPath executes tasks on hierarchical network components, which range from traditional data centers to core networks and fogs [186] to user terminal devices. From top to bottom, the device capability decreases, while the number of devices increases. Similar to CloudPath, SpanEdge optimizes the scheduling of computational tasks based on the communication latency and computational power of layered network devices processing the raw data.

- *At local edge and nearby edges:* The design of this type of edge systems is driven by the insight that mobile devices can benefit not only from the edge devices one hop away, but also the edge devices two, three, or even more hops away, with acceptable latency [174, 181].

2.2.4 Which devices provide edge resources?

Based on the answer to this question, edge systems can be divided into two major categories:

- *any available devices at the edge:* Examples include PCloud, Cloud-Sea, and FocusStack. One common feature shared by all these systems is that they need to efficiently discover collocated devices, and deploy executable code on them. PCloud forms a distributed resource pool to discover new resources and monitor resource changes. FocusStack involves a Geocast subsystem to find nearby devices and an OpenStack subsystem to deploy, execute, and manage containers on edge devices. To efficiently allocate tasks to devices, these systems also need to determine how the available device resources fit task requirements.

- *statically deployed devices at the edge:* Examples include CloudPath, SpanEdge, Precog, Cloudlet, and ParaDrop. One key feature of these systems is that they have no need to discover available edge devices at runtime. However, CloudPath, SpanEdge, and Precog operate under the assumption that edge resources may turn insufficient — if the edge servers lack resources to process a given task, these systems move the task further to the cloud for processing. In contrast, Cloudlet and ParaDrop (designed for multi-tenants) operate under the assumption that edge resources are sufficient to satisfy application requirements. In particular, Cloudlet assumes that edge servers have sufficient computing resources and stable power supplies.

2.2.5 What development model is used for edge applications?

In terms of the programming models, edge systems can be categorized into two major categories.

- *VM/Container based:* Examples include Cloudlet, ParaDrop, and FocusStack. Cloudlet enables developers to configure and deploy the service VM that contains the server code on the cloudlet, so that it is ready to be used by the clients. ParaDrop employs lightweight containers for resource virtualization. FocusStack is also container-based. The VM/Container based model may require additional resources and setup time, but it provides better isolation, security, and privacy protection.

- *Service based:* Examples include CloudPath, PCloud, Cloud-Sea, AirBox, and Firework. These systems ship executable service code to devices on demand, and expose as services aggregated/orchestrated microservices provided by multiple devices. Cloud-Sea is based on the existing RestFUL2.0 architecture, which could be too heavy-weight for IoT devices. AirBox enhances the security of edge-based execution by relying on hardware security mechanisms (e.g., Intel SGX).

Our own answers to the questions above lead to a design, in which an edge system provides on-demand computing, sensing, and communication capabilities to nearby mobile clients. To realize our design, we explore how all available devices at the edge can be engaged in computational scenarios. This dissertation research takes into account the possibility of device mobility, insufficient processing capacity, and power supply. Recognizing the dissimilarity and heterogeneity of edge devices and resources, the overriding goal of this research is to provide reliable, efficient, and consistent edge services under the above conditions.

Chapter 3

Self-Organized Edge Systems

The modern computing landscape is marked by several rapidly evolving realities. A typical user owns multiple mobile devices that differ in their types, platforms, and capabilities. For example, a user may simultaneously own a smartphone, a tablet, an e-reader, each of which runs a different operating system and offers vastly dissimilar processing capabilities, sensory functionalities, and networking interfaces. Furthermore, the number and variety of mobile devices in a typical household is even greater. Finally, the rapid developments in wearable computing and the Internet of Things (IoT) have the potential to increase these numbers for a typical user by as much as an order of magnitude in the near future.

Mobile devices have traditionally used the cloud as a means of enhancing their execution [85, 86, 160], both to improve the quality of service and to provide novel functionalities. However, accessing cloud-based resources could be infeasible, unsafe, and inefficient. On the other hand, with the rapid growth of the capacity and amount of mobile devices, the computational power could be provided by nearby mobile devices instead. All these scenarios give rise to the potential of leveraging nearby mobile devices, often owned by the same user or a community of users, as an alternative means of gaining additional resources.

Figures 3.1, 3.2 and 3.3 depict three scenarios exemplifying the conditions described above. In Figure 3.1, a smartphone application needs to search for a given face from all photos in the phone's album. Facial recognition is known to be computation/energy-intensive thus causing high latency/battery consumption, especially when the user has hundreds of photos.

In Figure 3.2, the driver is navigated through a smart glasses interface. However, keeping the glasses' GPS module on continuously could drain the device's battery quickly. In Figure 3.3, a smartphone user on a short-term trip to a foreign country needs to access the Internet. However, without a local mobile account, the phone cannot access any mobile data services provided by the available cellular network providers.

Although the users mentioned above are all short of either computational resources, context-

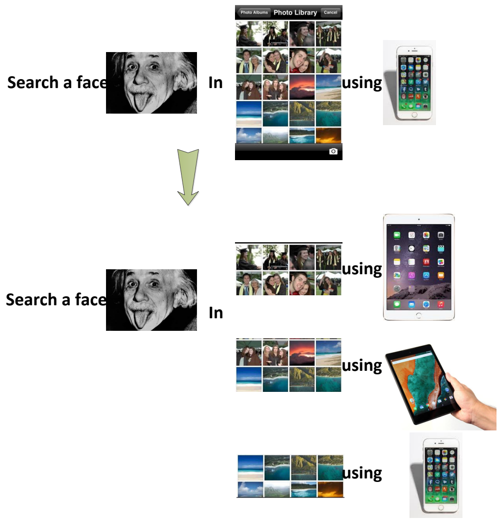


Figure 3.1: Scenario 1: Photo Recognition

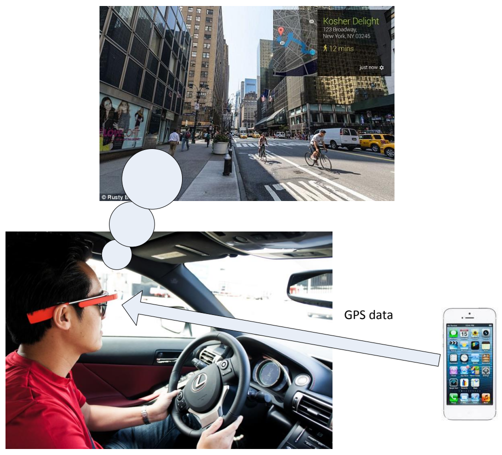


Figure 3.2: Scenario 2: GPS Sharing

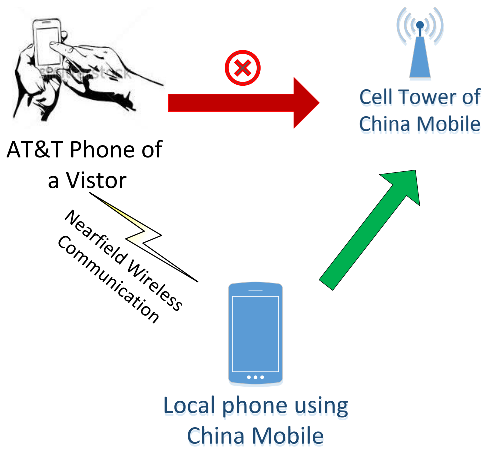

AT&T Phone of a Vistor
Cell Tower of China Mobile
Nearfield Wireless Communication
Local phone using China Mobile

Figure 3.3: Scenario 3: Data Plan Sharing

related resources, or network resources, various mobile devices (e.g., tablets, e-readers, and wearables, etc.), owned by themselves or their acquaintances may be in the immediate vicinity. These devices could provide the external resources required to solve the problems above. One could rewrite the mobile applications, so as to enable them to take advantage of such external resources. In scenario 1, one can reduce the execution time of computationally intensive tasks if they are run in a piecemeal fashion on nearby devices. In scenario 2, one can request GPS sensory reading from a nearby mobile device with larger battery capacity. In scenario 3, one can access the Internet by using a nearby mobile device, with a local mobile data plan, as a proxy that forwards the network requests and responses.

The aforementioned scenarios demonstrate how by sharing the resources of nearby devices, mobile applications can not only improve their quality of service, but also provide new functionality. However, several conceptual obstacles stand in the way of such resource sharing across heterogeneous mobile devices. For example, in scenario 1, one cannot execute offloaded mobile functionality on a different platform (e.g., running Android code on iOS). In scenario 2, one needs to be able to dynamically locate a nearby mobile device, whose battery capacity can accommodate long-lasting GPS sensor reading. In scenario 3, the programming interface to another user's mobile device must provide access to the device's voluntarily shared resources, while preventing misuse. The runtime in all scenarios must properly adapt to the mobility of the devices involved, ensuring efficiency and robustness.

In this chapter, we introduce two dissimilar approaches for supporting mobile edge computing, and compare their performance characteristics. In Section 3.1, we present our design of a mobile service market, a novel component for distributed edge computing systems that delivers the required executable code to edge devices on demand. In Section 3.2, we present a domain-specific language and its runtime system for decentralized P2P communication at the edge. In Section 3.3, we present and evaluate our system architecture for gateway-based edge computing.

3.1 Mobile Service Market

In mobile edge computing, a mobile or IoT device requests a nearby device to execute some functionality and return back the results. However, the executable code must either be pre-installed on the nearby device or be transferred from the requester device, reducing the utility or safety of device-to-device computing, respectively. To address this problem, we present a micro-service middleware that executes services on nearby mobile devices, with a trusted middleman distributing executable code. Our solution comprises (1) a trusted store of vetted mobile services, self-contained executable modules, downloaded to devices and invoked at runtime; and (2) a middleware system that matches service requirements to available devices to orchestrate the device-to-device communication. Our experiments show that our solution (1) enables executing mobile services on nearby devices, without requiring a device to receive executable code from an untrusted party; (2) supports mobile edge computing in practical settings, increasing performance and decreasing energy consumption; (3) reduces the mobile development workload by reusing services.

3.1.1 System Design

Micro-mobile services[1] extends the notion of the Service Oriented Architecture (SOA) for the needs of ad-hoc mobile execution on nearby devices. Unlike a traditional service that is hosted at a location identified by a fixed domain name (e.g., an IP address), mobile services reside at a mobile service market and deployed on devices for execution on demand at runtime. For a mobile service, its developers are supposed to provide several equivalent implementations for different targeting platforms. Since mobile services are expected to be executed on devices with limited resources, service developers have to design their solution with resource scarcity in mind. The acceptance criteria for hosting a service in a Mobile Service Market must be necessarily more stringent than those for accepting mobile applications to application markets.

As shown in Fig. 3.4, our solution contains three novel components: (1) The Mobile Service Market (MSM): an online service market for hosting and deploying mobile services; (2) Service Middleware that matches services with suitable devices at runtime as well as manages the communication across heterogeneous devices; (3) Programming model that provides a convenient interface for mobile application developers to choose the needed services and configure their requirements. We next describe the novel parts of our contribution in turn.

[1]For brevity, in the rest of the presentation we shall use the terms *micro-mobile services* and *mobile services* interchangeably.

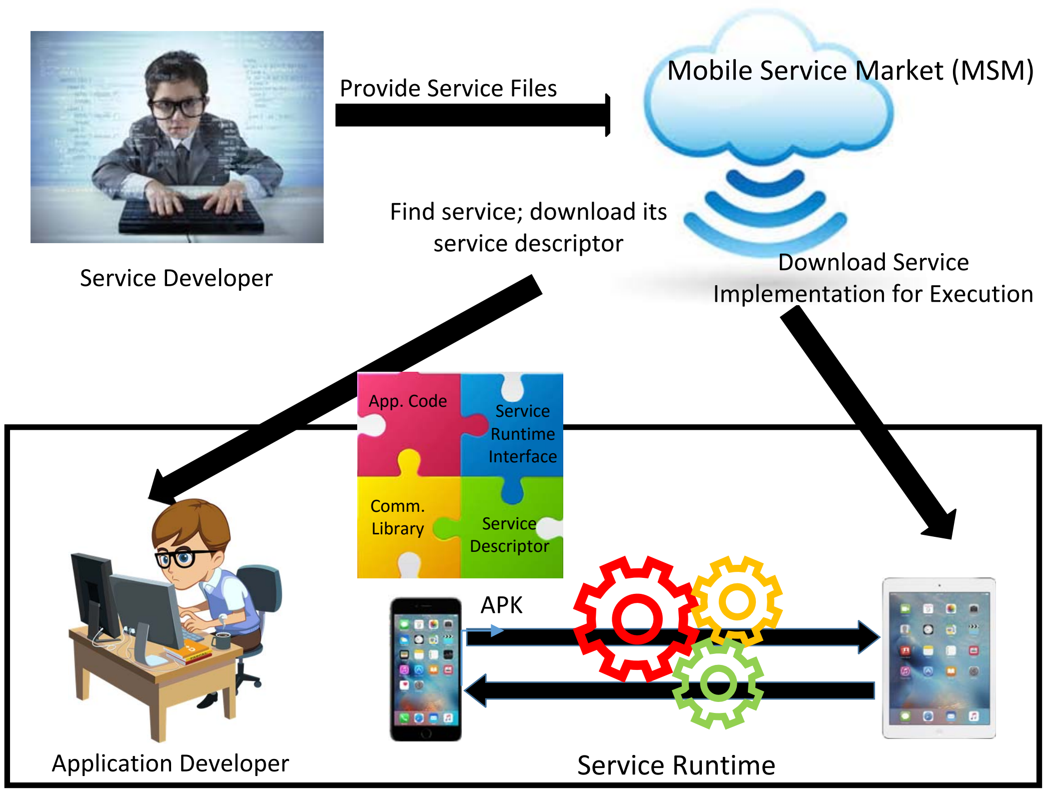

Provide Service Files
Mobile Service Market (MSM)
Find service; download its service descriptor
Download Service Implementation for Execution
Service Developer
App. Code
Service Runtime Interface
Comm. Library
Service Descriptor
APK
Application Developer
Service Runtime

Figure 3.4: Proposed MSM Architecture.

3.1.2 Service Market

Figure 3.4 shows the main components of the MSM architecture that codifies interactions across three different roles: service developer, application developer, and mobile user.

The mobile service developer identifies those pieces of application functionality that can be represented as mobile services. A service submitted to a service market must adhere to a format detailed below.

The application developer incorporates mobile services into their applications. To that end, they need to browse through the catalogs of mobile services of the MSMs that their users are likely to trust. They select the services that solve the resource scarcity problem at hand, and are able to change the constraints of the invoked services for the specific needs of their applications.

The mobile user will need to configure their device to be willing to accept the execution of services, dynamically downloadable from a given MSM. The middleware will manage the service's lifecycle, such as obtaining the latest version of the service execution package for further invocations.

In essence, the MSM combines the features of application markets and service repositories. Following the application market model makes it possible for users to rely on the reputation of a given market to have enough trust to allow the automatic installation and execution of such mobile services. At the same time, service developers would have to comply with the service market requirements, which likely would have to be more stringent than those of application markets.

Service Representation

Our design of MSM defines a typical mobile service as a collection of three elements:

Service Description: uniquely identifies a service as a combination of service name, version, usage scope, and parameters.

Service Execution Package (SEP): is a self-contained executable package that can be downloaded from the service market and executed on a mobile device. A service can be designed for one particular platform, several platforms, or all platforms by means of JavaScript execution. Platform availability is one of the selection criteria that mobile developers need to consider when deciding to use a mobile service in their applications.

Constraints: are requirements on the device that can be selected by the runtime to execute a service. Constraints are defined by service developers, with some of their parameters configured by application developers for the needs of a given application. Currently, our

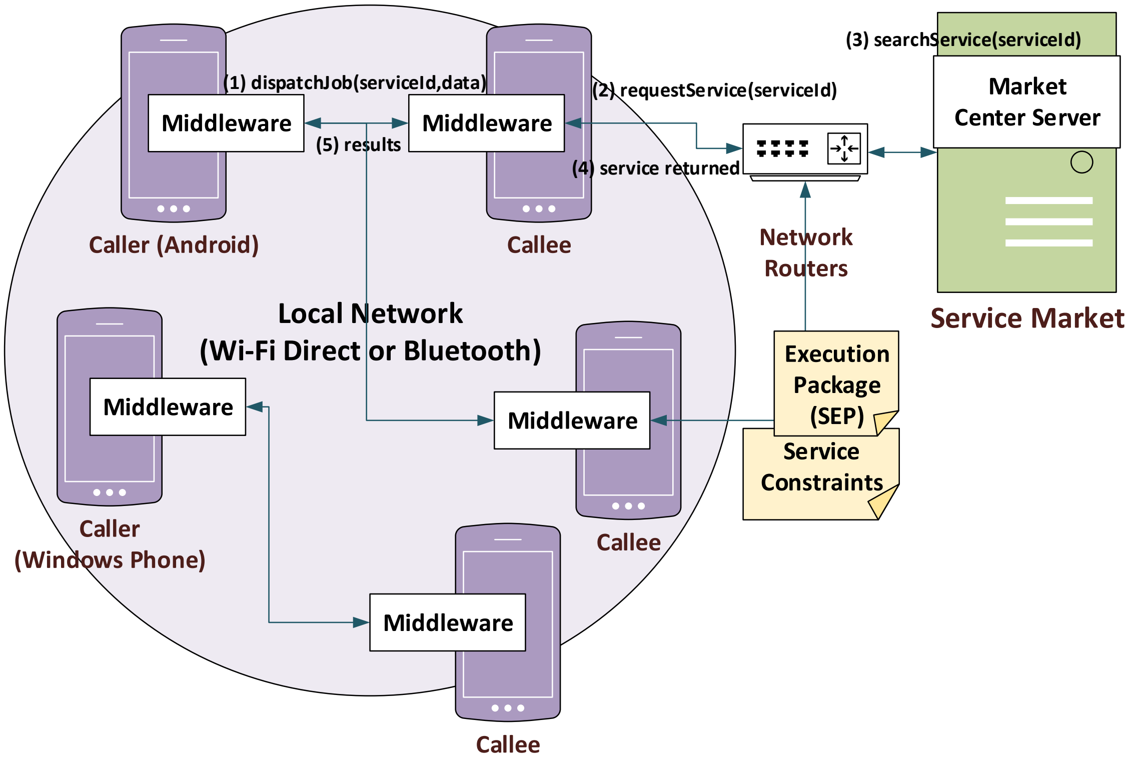


Figure 3.5: Service Market overview

reference implementation makes use of the following constraints:

- Sensor availability: `REQ` or `NREQ`. For example, a service may require a GPS sensor for execution.
- Battery threshold: `N(%)` – device does not respond to a service execution request if the remaining battery level is lower than $N\%$.
- Expected QoS Level: `(N)` – the detailed definition of expected QoS level (EQS) will be given in the next subsection. Generally speaking, EQS is a metrics of a mobile device's resource status.
- Network availability: `HIGH`, `LOW`, or `N.A.`
- Number of required devices: `N`, the number of required mobile devices to execute a service (e.g, $N > 1$: collaborative execution)

3.1.3 Service Execution Model (Middleware)

The middleware system provides a communication infrastructure for mobile devices, coordinating the execution of services between clients and servers. In this section, we explain each component of the middleware system in turn.

Discovering Available Nearby Devices

To discover available devices, a client device first sends out a peer discovery message to nearby devices, and then each device replies with its availability that represents resource capacities. The following JSON format shows a response message to the peer discovery message.

```
JSON : DeviceInfo {
  "availability": Boolean, "gps": Boolean,
  "network": ["low"|"high"|"off"], "EQS": Double}
```

where `availability` indicates whether the device is ready for any execution; `gps` shows the availability of GPS; `network` shows the network state, which is either `high`, `low` or `off`; and `EQS` is used for expressing the level of service execution capacity, which will determine service quality.

Selecting Available Devices

The middleware system's ability to select the most suitable devices is crucial for ensuring low energy consumption and high performance. To address this problem, we introduce a quality- and constraints-based peer selection mechanism that works as follows. First, after collecting `EQS` values from nearby devices, the client updates these values with the latest latency information. Then, if a service requires only one device, the middleware system selects the device that has the largest `EQS` value and meets the other selection criteria. Otherwise, it selects a number of suitable devices following the descending order of `EQS` values.

Once the client device has collected all the response messages from nearby devices, the most favorable devices for the given service execution are selected in accordance with the `EQS` value defined in the previous section. The `EQS` value is computed using the resource usage information including CPU, memory, battery, and network.

Service Execution and Fault Handling

The service execution procedure contains the following steps: (1)the client send the service execution request to the selected server devices; (2) the server devicess download the service execution package from MSM, and execute the SEP; (3) the server devices send the service execution results back to the client.

Due to the volatile nature of mobile networks, failures are a constant presence of mobile execution. As a failure handling strategy for mobile service execution, the Service Middleware on the client listens to the network-related updates. When there are any failures reported, the middleware, if possible, will attempt to by resume the service execution locally at the client for any reported failures. A simple checksum mechanism is used to verify the integrity of the execution results.

3.1.4 Development Support for Mobile Application Developers

To become a pragmatic solution to the resource scarcity problem of mobile devices, device-to-device mobile services must provide a convenient programming model to the mobile developers for them to invoke mobile services in their applications. In our reference implementation, we experimented with integrating the notion of mobile services with a modern Integrated

Development Environment (To support custom tools, modern IDEs offer an extensibility mechanism realized as plug-ins). The provided IDE plug-in provides three basic functionalities: searching for mobile services, generating sample code for invoking mobile services, and specifying service constraints.

Fig. 3.6(a) shows the main menu of our plugin, which provides three basic functionalities: searching for services, generating sample code for invoking services, and specifying service constraints.

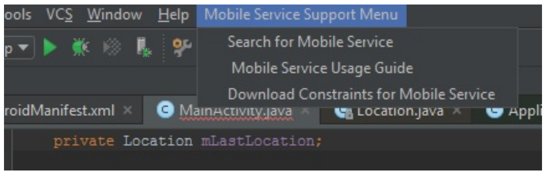
Fig. 3.6(b) shows the search panel for mobile services, through which application developers can search for the required functionalities based on keywords. Upon submitting a search query, the plugin connects to the MSM repository, obtains a list of all services, and filters those ones that match the specified keywords. The developer can further drill down into the returned services to learn about their version, usage scope and parameter sets.

Fig. 3.6(c) shows an example of generated service invocation code. This generated code snippet can be copied and pasted into the application project with minimal adjustments.

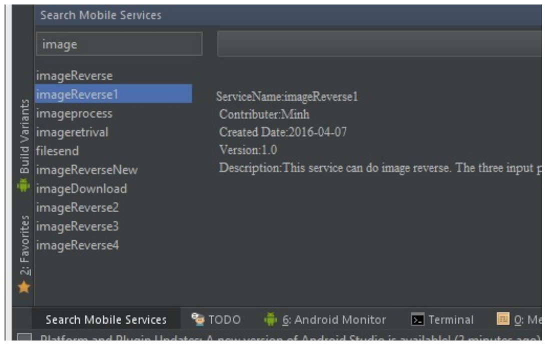
Fig. 3.6(d) shows the procedure of obtaining and modifying service constraints. The plugin downloads the service constraint specification for a given service and displays the constraints in a panel shown. The developer can specify the values of the constraints on the right side of the constraints view.

3.2 Resource Query Language: A P2P Approach

In this section, we present solutions that address the deep, conceptual challenges of enabling mobile devices to provide/use resources for/of nearby heterogeneous mobile devices. These solutions embrace heterogeneity, working with any pair of mobile devices, irrespective of their

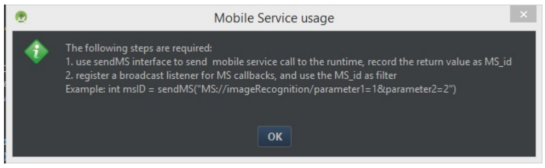

ools VCS Window Help Mobile Service Support Menu
Search for Mobile Service
Mobile Service Usage Guide
Download Constraints for Mobile Service
roidManifest.xml
MainActivity.java
Location.java
private Location mLastLocation;

(a) Menu of IDE-plugin supporting mobile services



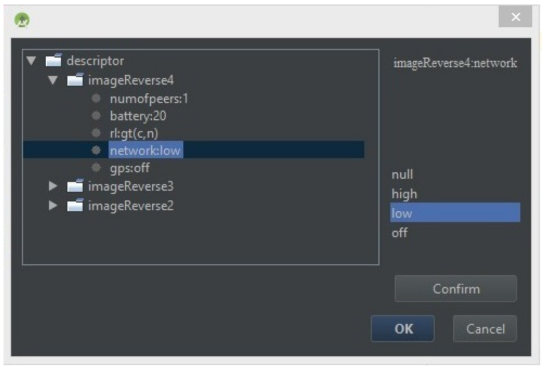
ServiceName:imageReverse1
Contributer:Minh
Created Date:2016-04-07
Version:1.0
Description:This service can do image reverse. The three input

(b) Searching a mobile service in MSM


Mobile Service usage

The following steps are required:
1. use sendMS interface to send mobile service call to the runtime, record the return value as MS_id
2. register a broadcast listener for MS callbacks, and use the MS_id as filter
Example: int msID = sendMS("MS://imageRecognition/parameter1=1&parameter2=2")

(c) Mobile service usage guideline.



numofpeers:1
battery:20
rl:gt(c,n)
network:low
gps:off

imageReverse4:network

(d) Traversing a project to detect invoked mobile services.

Figure 3.6: Screen shots of the IDE-plugin support

platforms, operating systems, or installed applications. Also, the presented solutions reduce the programmer's effort in creating reliable and efficient functionality for sharing resources. This work makes the following contributions:

- We study and reveal how existing applications can benefit from shared resources of nearby devices.
- We design the Resource Query Language (RQL)—a declarative domain-specific language for accessing shared resources of nearby devices. RQL makes it possible to declaratively express resource sharing requests by simply specifying the preferred devices, resource types, and the actions to be carried out. The RQL runtime is designed

with provisions for energy efficiency, latency optimization, and privacy preservation when executing across heterogeneous mobile devices.

- We provide a reference implementation of the RQL language and runtime support on major mobile platforms, including iOS and Android. We also describe example applications that make use of RQL to access resources across the iOS and Android platforms.
- We evaluate the programmability and efficiency of our technical approach through a case study and experiments. Our results indicate that the presented solutions can improve the productivity of mobile programmers, as well as improve the performance/energy efficiency of mobile applications.

3.2.1 RQL Design

RQL is a platform-independent, domain-specific language that enables heterogeneous devices to seamlessly share their resources. We designed RQL around the RESTful architecture [42], a proven solution for many of the complexities of engineering dynamic, heterogeneous distributed systems, including the WWW.

In our target domain, we leverage the flexibility of this architecture to hide the complexity of the inherent heterogeneity of mobile devices that need to participate in device-to-device resource sharing scenarios. We observe that in this domain, the actual operations on the shared resources are limited to a small set, and exploit this observation to provide a concise yet powerful DSL for resource sharing. Specifically, the design of RQL follows the verb/nouns paradigm: nouns express the requested resources, while verbs express the actions performed on these resources.

We next present RQL by example. Consider an RQL statement: `pull glass:sensor/orientation`.

This statement will retrieve the readings of the orientation sensor of a glass device, if it happens to be in the vicinity; it will return a `null` reading otherwise. The specific details of locating a glass device, connecting to it, retrieving its readings, etc. are handled by the RQL runtime.

Nouns: RQL represents the resource intention with nouns. Specifically, the nouns comprise the following parts: "device description:resource description/specific name".

Device description defines device types (e.g., glass, smartphone, tablet, etc.) or specific characteristics (e.g., name, owner, OS, etc.). Resource description defines the type of resource (e.g., sensors, services, files, etc.) followed by specific names (e.g., sensor/orientation, sensor/gps, service/facerecognition, service/httpsend, etc.).

Verbs: Following the RESTful design principles, a small number of verbs manipulates an infinite number of nouns. In particular, RQL defines only four verbs: `pull`, `push`, `delegate`, and `bind`. As shown in Fig. 3.7, "`pull`" retrieves data from the service interface of another device immediately; "`push`" sends data from the source device to the target device; "`delegate`" sends some parameters and then gets the execution results back; finally, "`bind`" establishes a persistent connection to a device to obtain the value changes of a specific sensor.

Adverbs: Although traditional RESTful interfaces consists of only verbs and nouns, RQL integrates adverbs as informed by some prior research on fault-tolerant RESTful services [40]. In RQL, adverbs can express how commands should be executed in terms of time or quality constraints. For example, an adverb can express the timeout value for a `pull` command (in ms) (e.g., `pull external:alg/OCR -latency < 500ms`). Another adverb is `-blocking` (e.g., `pull external:sensor/GPS -blocking`, which expresses that the RQL call to retrieve the GPS reading should block, to return only when a GPS reading becomes available or the call has failed. By default, all RQL statements are non-blocking with the

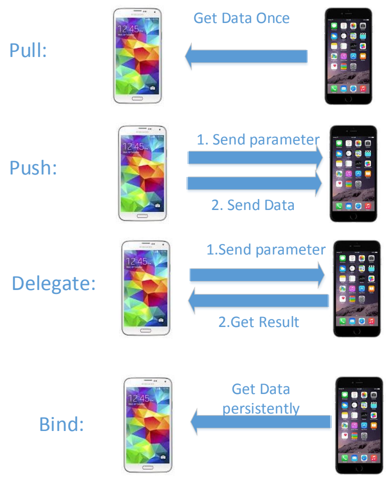


Figure 3.7: Defined RQL Verbs

results communicated via an asynchronous callback mechanism. We discuss a programming scenario involving the `-blocking` adverb in Section 3.2.3.

Fig. 3.8 depicts several examples of using RQL. The first example is concerned with getting GPS readings from another device. The second example sends a data file to a remote device (belonging to user John) to use as a parameter to a facial recognition algorithm. The third example directs a remote device to perform an HTTP request for a given URL and sends back the obtained output. The fourth example establishes a persistent connection to get orientation sensor updates from John's smart glasses device.

Sometimes the source device may need to execute a sequence of RQL statements on the same target device consecutively. To that end, RQL features the "|" binary operator, which specifies that its operands are to be transmitted in bulk to the target device and executed in sequence. Consider the source device needing to execute both the OCR and language translation algorithms one after another on the same target device. The programmer can express this functionality in RQL as shown in line 2 of Fig. 3.8. Batching RQL requests may also reduce their aggregate latency.

```
1. pull any:sensor/GPS
2. push John:file/myphoto.jpg -i /DCIM/20170319323.jpg | delegate service/
   faceRecognition
3. delegate any:service/http -t http://www.google.com
4. bind John/glasses:sensor/orientation
```

Figure 3.8: RQL Examples

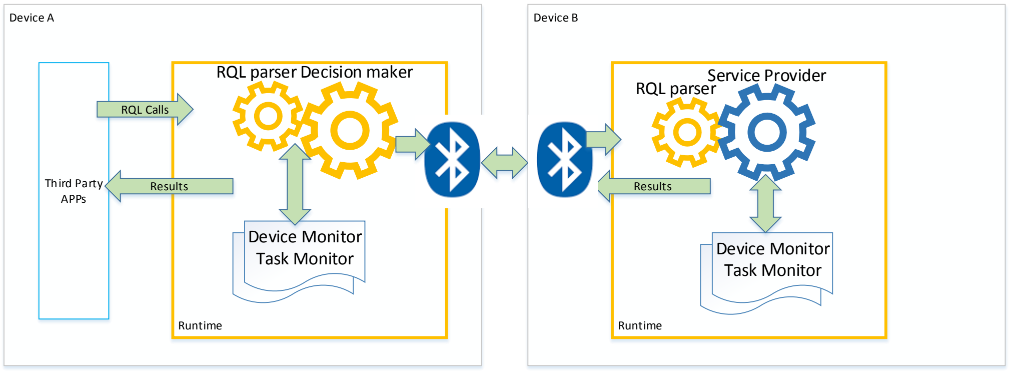


Figure 3.9: General Design of Runtime Support

3.2.2 Runtime Design

To meet its design goals, RQL requires sophisticated runtime support for mainstream platforms (i.e., Android, iOS, and Windows Phone). In this section, we identify the requirements and outline design of such runtime support. With respect to requirements, the RQL runtime must reconcile the need for efficiency with that of portability and ease of implementation. Hence, we have deliberately constrained our runtime design to the application space, so as to avoid low-level, platform-specific system changes. In other words, the user should be able to install the RQL runtime as if it were a regular mobile application, albeit with extended permissions (e.g., access to all sensors, the ability to connect to remote services via all available network interfaces, access to local application data and external storage, etc.)

The runtime support, whose basic flow appears in Figure 3.9, includes three basic modules: client, server, and monitor. The client module of Device A accepts an RQL request and determines whether the request can be executed by a nearby device (Device B) by querying a distributed registry of nearby devices and resources they provide. The devices communicate by means of near field communication interface (e.g., Bluetooth). The server module of Device B parses the request, executes it, and returns the result back to the client module of Device A. The monitor module comprises two parts: device and service status. The device status monitor keeps track of the battery levels, resource usage status, and locations of nearby devices. The service status module monitors the energy consumption/latency of the services provided by the nearby devices.

Choosing Communication Channels: In our runtime design, Bluetooth Low Energy (BTLE) serves as the major communication mechanism for two reasons: 1) BTLE is known to be the most energy efficient way to discover/announce external services. Although WiFi and Bluetooth are popular device-to-device communication mechanisms, their energy consumption levels are larger than that of BTLE, both in active and idle modes; 2) to support heterogeneity, the runtime must be able to use a communication mechanism supported by major mobile platforms. Mainstream mobile communication mechanisms, including WiFi-direct and traditional Bluetooth, cannot connect a recent (i.e., 4.4.2 and up) Android device with an iOS device.

However, BTLE does have some limitations. Chief among them is the primary use-case for BTLE: command transmission and small data-size transmissions. The main purpose of BTLE is to send small bursts of data for extended periods of time while consuming minimal energy. The largest size package BTLE will send is 20 bytes. Therefore, when the runtime needs to send a data file to another device, using a different communication mechanism can provide performance advantages.

To overcome the limitations of BTLE when transferring larger data volumes, our design includes an optimization that makes use of edge servers. When transferring a data file, the runtime at the source device uploads the file to an edge server, and send the URL of that file to the target device via a BTLE connection for the target device to download. Nevertheless, it is worth noting that, with both Android and iOS constantly improving the relatively new inter-device communication mechanisms, our runtime is capable of communicating via WiFi-direct, once it becomes available for heterogeneous devices.

Choosing Target Device: When multiple devices can be used for a given task, selecting the correct device could save the overall energy consumption of all devices. For tasks that require the service to send HTTP requests, as the 3G chips would still cost energy when the data transmission is finished, combining multiple requests and sending them at once could greatly save the overall energy consumption. For tasks that require a specific sensory reading like GPS, the major energy consumption happens when the target device tries to obtain the sensory reading. Therefore, combining multiple sensory requirement tasks to the same device could also reduce the overall energy consumption.

We intend to use an incentive strategy to encourage batching HTTP requests and sensory data requests to the same target device. The basic idea is to let the device which has already been the delegation of such requests to ask for lower bid prices for other tasks of the same kind.

Reducing Latency: Different from the HTTP requests and sensory data requests, RQL requests which need to perform computationally intensive tasks can not be energy-optimized by being batched to a same delegation. On the contrary, when such tasks are combined to the same target device, their time/latency usually gets larger. Therefore, in the runtime, for those RQL requests that want to process an amount of computation intensive tasks through multiple devices, the runtime needs to act as the load balancer: it needs to divide the necessary tasks into chunks between multiple devices in a way that the overall waiting time is minimized.

The most accurate way to balance loads across numerous available devices is to get real-time loads from each devices and also the execution time of each task in advance. However, the frequent communication among devices costs extra energy and clogs the channel as it is occupied for a larger amount of time. In such cases, the solution we take is to log the load of each device in the format of how many tasks are running or waiting. The running tasks of surrounding devices are updated through the device monitor's scanning action. When the runtime assigns one task to a device, the device's load of is incremented by one; when it get the result back from a device, its load is decremented by one. Therefore, each time when the runtime needs to assign a task, it assigns it to one of all the devices providing that service with the lowest load.

Incentive Strategy In the presence of multiple unrelated mobile users, the adopters of this technology may face the problem of having to motivate them to share the resources of their devices. One possible approach is putting in place an incentive strategy that employs micro-transactions for devices to pay for the external resources consumed. The payments can be represented as marketplace credits to pay for using shared resources in the future or even as a standard currency.

The RQL runtime's design includes an incentive strategy based on the reversed auction model, as shown in Fig. 3.10. Before a device can issue an RQL request, the runtime scans all nearby devices and gets their bid prices for the required service. It then chooses a device with the lowest bid price as the target of offloading. When other devices have the same bid prices, it randomly picks one, or chooses one according to their loads. After the task is finished and the results are returned, it pays the chosen device the bid price as incentive.

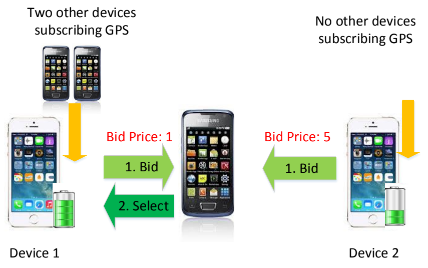

Two other devices subscribing GPS
No other devices subscribing GPS
Bid Price: 1
1. Bid
2. Select
Bid Price: 5
1. Bid
Device 1
Device 2

Figure 3.10: Flow of Reversed Auction

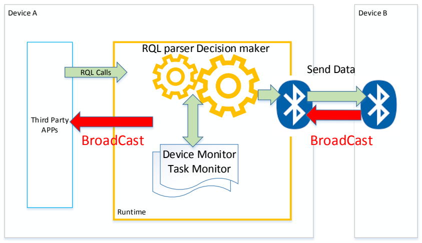


Figure 3.11: Possible Attacks

This strategy would help motivate unrelated users to make the resources of their devices available for sharing.

An incentive strategy can also take energy consumption into consideration when offering bids. For example, a mobile device already delegating HTTP or sensory tasks, should be able to offer lower bid prices than idle devices, as performing additional tasks would incur smaller energy costs. Therefore, the probability of forwarding the majority of HTTP requests or sensory reading tasks to the same device would increase. In such cases, the energy consumption of all the participating devices becomes minimized. Hence, the initial investment into recruiting mobile users to participate in resource sharing will be amortized by the future improvements in usability and performance. Incentive strategies thus constitute a promising future research direction for this work.

Privacy and Security Fig. 3.11 describe the potential threat of privacy leakage and security issues, where device A and device B are the source and the target, respectively. The security threats could arise in the following scenarios: 1) When the runtime on device A broadcasts the result of some third party application, it could be wiretapped by a malware installed on that device. 2) when the runtime on device A receives the broadcast from device B through Bluetooth, another device C binding to the same Bluetooth channel might get that message as well. One can counter this security threat by encrypting the message. To solve the problem, the third party application will need to provide a public encryption key for each RQL request, so that the runtime can encrypt the result with that key. This way, it is only the third party application with the private key that can decrypt the result. Although our reference implementation does not yet include this security mechanism, our design makes it possible to straightforwardly add it to the runtime.

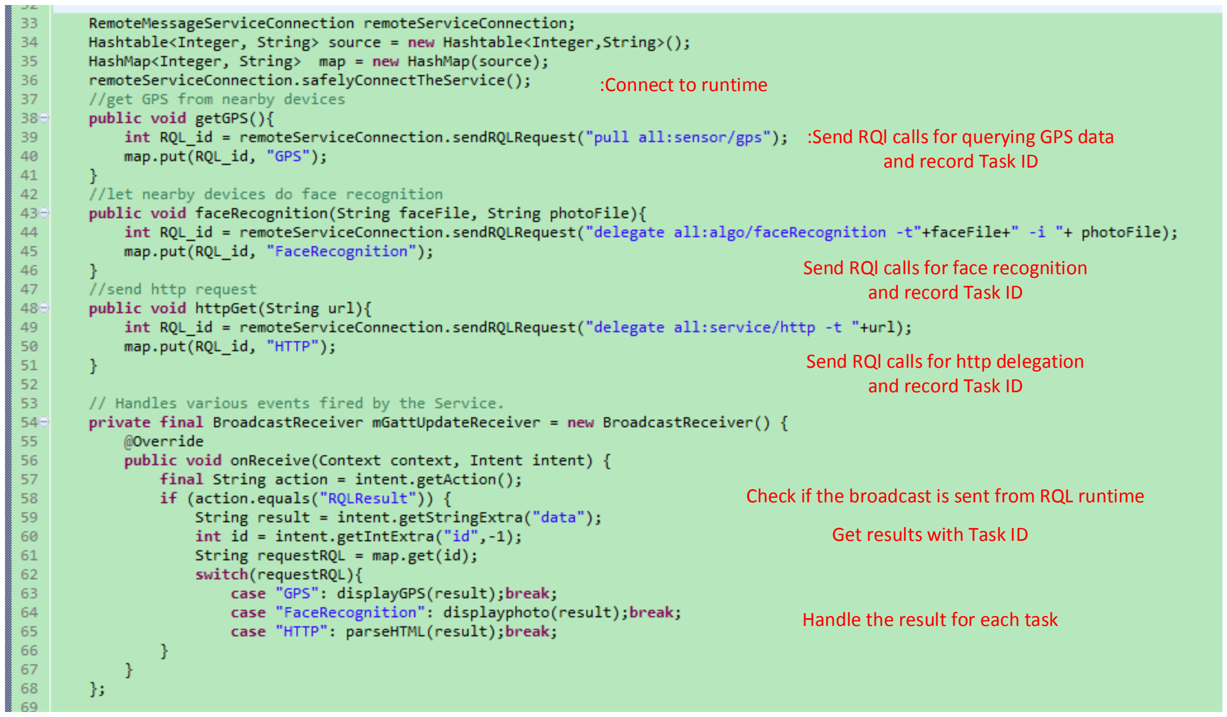
3.2.3 Reference Implementation

Our reference implementation of RQL and its runtime concretely reifies the design decisions we described in Sections 3.2.1 and 3.2.2. While we have implemented all the described features of RQL including the required runtime support, some of the optimization and privacy provisioning features of the runtime remain a work in progress.

To demonstrate our implementation, we next describe how we used it to address the resource sharing needs in the three motivating examples mentioned in the beginning of this chapter. The snippets of Java code in Fig 3.12 show how the three source devices use RQL to access resources of nearby target devices.

3.2.4 Evaluation

In this section, we describe how we evaluated various aspects of the reference implementation of RQL, detailed in Section 3.2.3. Our evaluation comprises a small user study,



```
    RemoteMessageServiceConnection remoteServiceConnection;
    Hashtable<Integer, String> source = new Hashtable<Integer,String>();
    HashMap<Integer, String>  map = new HashMap(source);
    remoteServiceConnection.safelyConnectTheService();
    //get GPS from nearby devices
    public void getGPS(){
        int RQL_id = remoteServiceConnection.sendRQLRequest("pull all:sensor/gps");
        map.put(RQL_id, "GPS");
    }
    //let nearby devices do face recognition
    public void faceRecognition(String faceFile, String photoFile){
        int RQL_id = remoteServiceConnection.sendRQLRequest("delegate all:algo/faceRecognition -t"+faceFile+" -i "+ photoFile);
        map.put(RQL_id, "FaceRecognition");
    }
    //send http request
    public void httpGet(String url){
        int RQL_id = remoteServiceConnection.sendRQLRequest("delegate all:service/http -t "+url);
        map.put(RQL_id, "HTTP");
    }

    // Handles various events fired by the Service.
    private final BroadcastReceiver mGattUpdateReceiver = new BroadcastReceiver() {
        @Override
        public void onReceive(Context context, Intent intent) {
            final String action = intent.getAction();
            if (action.equals("RQLResult")) {
                String result = intent.getStringExtra("data");
                int id = intent.getIntExtra("id",-1);
                String requestRQL = map.get(id);
                switch(requestRQL){
                    case "GPS": displayGPS(result);break;
                    case "FaceRecognition": displayphoto(result);break;
                    case "HTTP": parseHTML(result);break;
                }
            }
        }
    };
```

:Connect to runtime
:Send RQl calls for querying GPS data and record Task ID
Send RQl calls for face recognition and record Task ID
Send RQl calls for http delegation and record Task ID
Check if the broadcast is sent from RQL runtime
Get results with Task ID
Handle the result for each task

Figure 3.12: Mobile Application Code using RQL

Table 3.1: Lines of Code

	Runtime Based	Built from scratch
GPS request	20	370
HTTP request	20	556
Facial Recognition	32	883

various performance/energy efficiency micro-benchmarks, and a robustness assessment of our retrofitting approach.

Programmability First, we evaluated the software engineering benefits of our programming model. To that end, we compared two different implementations of the same resource-sharing scenario: original with all resource sharing functionality implemented from scratch and RQL-based with the major functionality provided by the RQL runtime. In Table 3.1, for each implementation, we report the total lines of uncommented code (ULOC).

As one can see, using RQL reduces the amount of code the programmer has to write by a factor ranging between 20 and 28. Considering that the written code involves complex asynchronous, distributed processing, this code size reduction is likely to have a high positive impact on the code quality.

To empirically assess how well RQL can assist the programmer in putting in place the inter-device resource sharing functionality, we conducted a user study. To that end, we recruited 10 Junior to Senior level Computer Science students from an intermediate Android development class at Virginia Tech. We divided the recruited students into 2 groups, the experimental and control groups, for novice and experienced Android developers, respectively. The experimental group comprised 6 students with no prior experience in Android programming, while the control group comprised 4 students with several years of Android development experience.

In the beginning, we briefly introduced the concepts of AIDL services, broadcast receivers, and Bluetooth LE. Then, each group was given 90 minutes to complete the programming task of obtaining the GPS sensor reading from an iOS device to an Android device. The experimental group was asked to use RQL, while the control group was asked to use any existing, mainstream Android API. The control group was also given an Android chat sample application as an example from which to draw device-to-device coding idioms.

Table 3.2 presents the results of the study. To our surprise, none of the students in the control group were able to complete the task successfully, which demonstrates the non-trivial nature of device-to-device communication. The results of the experimental group, armed with RQL, were mixed, with 3 students successfully completing the task, with the remaining 3 giving up before the experiment concluded. Because the group using RQL comprised non-experienced Android programmers, the results above indicate that our programming abstraction provide value by streamlining the process of implementing device-to-device interactions and can become a pragmatic tool for future applications.

Table 3.2: Study Results

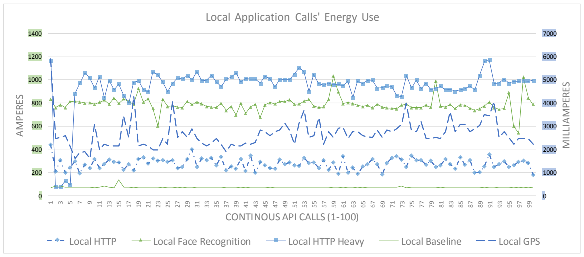
Group	1	2
Familiarity with Android Development	Beginner	Familiar
Number of students	6	4
Number of students completed the task	3	0

Table 3.3: Energy Consumption per Second

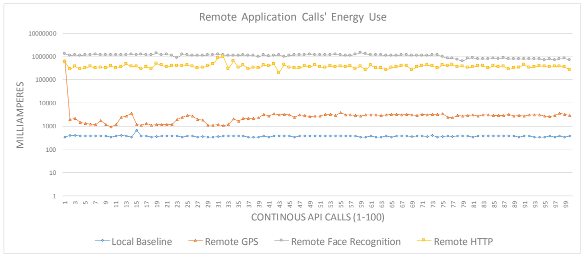
Status	Energy (mA)	Status	Energy (mA)
ScreenOn	100		
BluetoothOn	1	BluetoothActive	66
CpuIdle	92	CpuActive	242
WiFiOn	6	WiFiActive	102
GpsOn	60	GpsActive	300
3GOn	10	3GActive	250

Experiment Setup The hardware setup for the following experiments include 4 Android mobile devices (1.5GHz dual-core CPU, 2GB RAM) used as source devices, and 2 iOS devices (1 iPhone 6 and an iPad mini) used as target devices.

To evaluate the energy consumption of these devices, we recorded the execution time between "Start" and "Stop" tags, adding tags for actions, such as "Screen On", "Bluetooth On", "Bluetooth Active", "3G Active", "GPS Active", "CPU Idle/Active" etc. Table 3.3 shows the manufacturer provided values for energy consumption of these operations. For all graphs, we refer to 'local' and 'remote' meaning requests processed on the user's local device and some external nearby device, respectively.



(a) Local Energy Tests



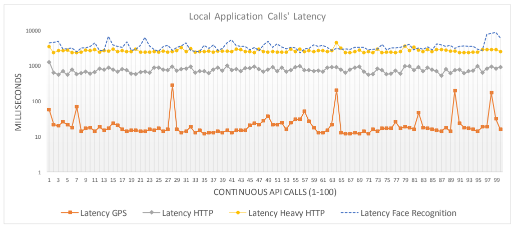
(b) Remote Energy Tests

Figure 3.13: Various local and remote RQL command energy usage

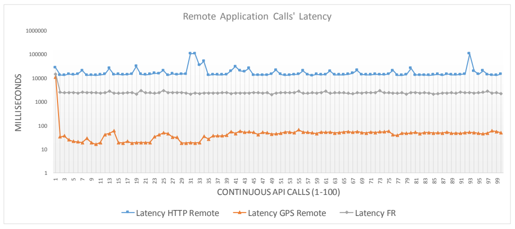
Local and Remote Energy Evaluation First, we examine the motivating examples' performance in terms of the energy usage in both the local API calls and the corresponding remote RQL calls. Figure 3.13 shows the energy used by 100 identical RQL requests on the same and across different devices, respectively. Because of the vastly different energy consumption levels between sensor data and heavy HTTP requests, we use both linear and logarithmic vertical scales to present the results.

The graphs show that, excluding some outliers, both local and remote RQL calls consume energy consistently throughout the experiments. The baseline of both figures is identical and essentially shows how an idle application would be consuming energy. In both local and remote calls, the GPS sensor retrieval consumes far less energy than either of HTTP requests or Facial recognition. To compare various protocols, we also benchmark a "Heavy" HTTP request, representative of work-intensive web-based processing. Given the extensible nature of the RQL runtime, one can easily add emerging communication mechanisms, which can outperform BTLE when executing heavy HTTP requests or other high-throughput processes.

Because communicating with nearby devices consumes additional energy, local RQL calls increase their energy efficiency when processing small loads of requests. However, for requests that can be distributed across several available devices, both energy costs and processing latencies decrease precipitously. Figure 3.13 also reveals cache correspondences between the same device, primarily for sensor data (GPS). Thus reading the GPS data incurs a single large, upfront cost of connecting to the device, but internally optimizes the subsequent request via the assumption that the GPS readings have not changed. This internal optimization explains the plummet in energy costs of accessing remote sensor data, such as GPS.



(a) Local Latency Tests



(b) Remote Latency Tests

Figure 3.14: Various local and remote request latency use

Local and Remote RQL Latency Experiments Consider Figure 3.14 that shows local and remote latency, respectively. These two graphs demonstrate an important practical advantage of accessing resource of nearby devices. When examining GPS, latency drops steeply similar to energy in the previous section, after incurring the upfront cost of connection. This amortization of initial connect requests ensures far better median latency for these remote calls. In fact, we see that for a computationally intensive operation, such as Facial Recognition, the latency is smaller in remote RQL calls by a factor of nearly 1.4 for only a small request size. If we consider sending large requests for Facial Recognition across even a small subset of nearby devices (say only 3 external devices), the resulting latency reduction far outweighs the additional energy use incurred across all devices in use.

Figure 3.15 presents a full comparison of median energy and latency measurements. This graph supports our initial assumption about the trade off in energy for decreased latency when processing various request types. It is clear that the only outlier is processing HTTP requests remotely. Given the nature of BTLE small packet transmission size restriction, we observe a larger latency since each piece of the HTTP request is broken up and sent individually. Referring back to one of our motivating examples, consider the traveler to a foreign country who is unable to access local mobile data towers. Providing this functionality to the end user is important irrespective of the resulting performance, as long as it is not prohibitively poor. In other words, not outperforming local requests is a minor hindrance in comparison with not being able to process any requests at all. Nevertheless, this limitation

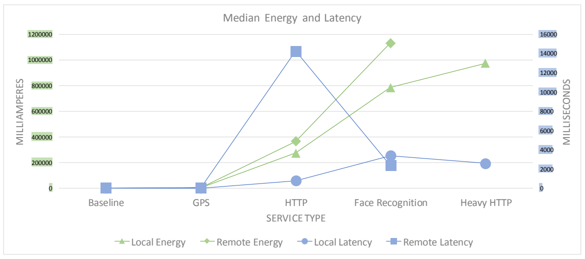


Figure 3.15: Median Energy and Latency Across Various Requests

of BTLE motivated our efforts to optimize the RQL runtime.

3.3 Programmable Mobile Device Cloud

Leveraging the resource capabilities (computation, storage, sensing, network, etc.) of co-located mobile devices at the edge of the network to execute tasks is generally referred to as *Mobile Device Computing (MDC)* [118]. In this section, we introduce a novel system architecture, based on microservices. Although known for their applications in cloud-based scenarios [169, 170], microservices also fit naturally for the MDC environments. Microservice architectures express application functionality as a collection of interacting micro functionalities, each represented and managed as an external service. Similarly, our architecture represents and manages remote functionalities as microservices, which can be invoked on demand. Further, our architecture delivers the executable packages to the available MDC devices by downloading them from a trustworthy microservice market.

In particular, our software architecture facilitates the process of finding the most suitable device to execute a microservice. Programming support is provided via a domain-specific language that makes it straightforward to express: 1) capabilities offered by the available MDC devices, 2) microservice demands and their non-functional requirements (NFRs) (e.g., latency, reliability, cost, or any other microservice-specific aspects). We also notice that it would be impossible to directly translate device capabilities into NFR satisfiability, without

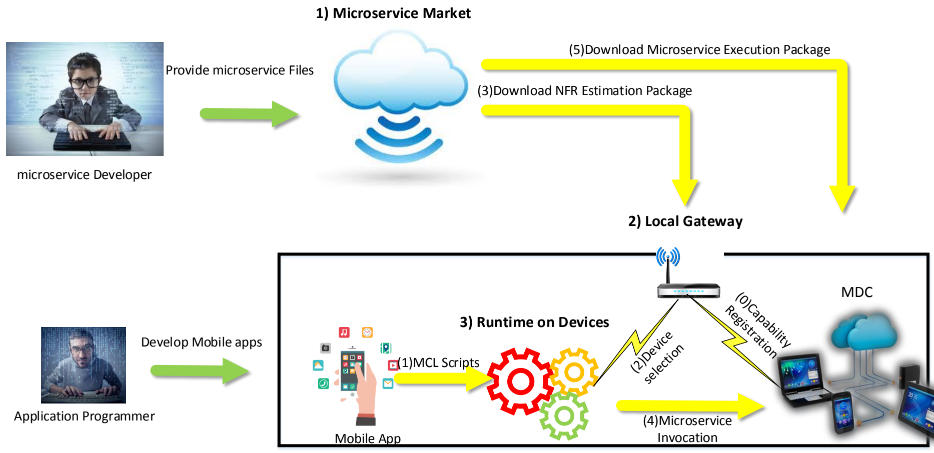


Figure 3.16: System Architecture Overview.

the domain-specific knowledge possessed by microservice developers. Hence, the architecture features a novel network component, the microservice gateway, responsible for collecting device capabilities in order to estimate how they satisfy the NFRs.

The major contribution of this work is three-fold:

- A microservice-based software architecture that lowers the barrier for mobile app developers to use MDCs.
- A domain-specific language and its distributed runtime for expressing and matching the application's functionality demand and the MDC resource supply.
- A realistic use case implementation and performance evaluation of the aforementioned architecture.

3.3.1 System Architecture

Our software architecture is supported by the system runtime, which comprises four parts (see Fig.3.16): 1) a client device that requests a functionality from MDC; 2) a local device that serves as gateway by maintaining an up-to-date mappings between the available MDC devices and their capacities; 3) a microservice market (MSM for short), a cloud-based repository that delivers the executable code of a given microservice; 4) a set of MDC devices that share their capabilities, as detailed next.

MicroService Market (MSM) MSM(see Section 3.1) combines features of application markets and service repositories. Following the application market model enables devices to automatically download and execute the required microservices, while following the service repository model enables application developers of the client apps to implement the required functionalities as microservice invocations, to be executed by MDC devices.

In the original design of MSM, a mobile device must download the microservices before it can be allocated to provide them. The devices are responsible for estimating their fitness to satisfy the NFRs of a given task and report the results to the gateway. By contrast, our new design enables the gateway to estimate how well the available devices can satisfy a task's NFRs, prior to deploying any microservices.

Local Gateways A typical cloud-based microservice architecture features a centralized service registry, a collection of registered device-to-microservice mappings, with a remote interface through which clients can bind themselves to the microservices they want to invoke. Notice that MDC applications need to invoke microservices on the devices reachable via short-range communication methods (e.g., WiFi, Bluetooth), rendering cloud-based registries inapplicable.

Hence, our system architecture features a novel system component: a local gateway that replaces the standard cloud-based service registries. Each mobile device cloud should have a local gateway that could be either a stationary device, connected to a permanent power supply, or a battery-operated mobile device. Unlike its cloud-based counterparts, local gateways maintain a registry of available device capacities of the MDC, instead of the microservices provided by the devices.

Runtime Support on Devices The runtime runs as a regular mobile app on the server and client devices. In general, the runtime accepts an MCL script to execute, either from the application via inter component communication (ICC), or from other devices via socket-based HTTP requests. On an MDC device, the programmer can specify the capability to share by interacting with the device's runtime using an MCL script. On a functionality demanding device, an app can first find the MDC device by querying the local gateway using an MCL script, and then invoke the microservice on the MDC device by passing it an MCL script with execution parameters.

Execution Flow Fig.3.16 also introduces our system architecture's execution flow. The mobile devices periodically register their shared capabilities to the connected gateway (step 0). When a mobile app requires to execute a microservice on MDC, it first sends MCL scripts to the runtime on the client (step 1). The runtime then interacts with the reachable gateway in its vicinity, to query the most suitable device for microservice execution (step 2). The gateway downloads the NFR estimation algorithm of the required microservice from MSM (step 3), applies it to select the most suitable MDC device(s), and sends the connectivity information of the selected devices back to the client. Then, the client connects to the selected device to initialize the microservice execution (step 4). The selected device downloads the execution package from MSM, and sends the execution results back to the client (step 5).

```
<MCL Script> ::= <Action> <Target> <Parameters>
<Action> ::= "reg"|"stop"|"query"|"exec"
<Target> ::= {<Resource> ","}+ | <Microservice>
<Resource> ::= "network"|"compute"|"sensor/"<Sensor>
  <Sensor> ::= "GPS"|"Cam"|"Mic"|"Motion"|"Light"|String
<Microservice> ::= String
<Parameters> ::= <Lease>|<Device Selection>|<Execution Param>
  <Lease> ::= "-t=" Numeric "-c=" Numeric
  <Device Selection> ::= ["-n=" Numeric]["-h="String]["-l="String]
  <Execution Param> ::= [String "=" String|Numeric]+
```

Figure 3.17: MCL EBNF Definition.

3.3.2 MCL Definition and Use Case

We first introduce the grammar of MCL, and explain its semantics for expressing the supply of device capabilities and the demand for microservices.

Functional Requirement We first summarize what functions MCL provides:

1. Specify device capability to share: The MDC devices need to specify what capabilities to share.

2. Find device for executing a microservice: The functionality demanding device needs to obtain one or more MDC devices, whose capabilities 1) fulfill the general execution requirements of a microservice (e.g., in use case 2, taking picture requires the device to share camera), and 2) best satisfy the NFRs (e.g., in use case 1, the app prefers an MDC device that can finish facial recognition most quickly).

3. Execute a microservice on a device: The functionality demanding device can start microservice execution on a selected MDC device.

Grammar Definition An MCL script comprises three parts: `Action`, `Target`, and `Parameters`. `Action` stands for the method, which includes (1) register device capabilities, and remove the registered information (`reg/stop`), (2) query microservice provisioning (`query`), and (3) execute microservice (`exec`). The `Target` can be either `Resources` (for `reg` and `Stop`), or `Microservice` (for `query` and `Exec`). The `Resources` includes network, computing, and sensors (e.g., GPS, camera, microphone, motion sensors, light sensors, etc.). `Microservice` is a string representing a unique ID of the related microservice function (e.g., "faceReco").

`Parameters` describes the action. When registering device capabilities, MCL enables specifying the leasing time (`-t`, for how long the capabilities will still be available), the incentive multiplier (`-c`, to be used to calculate the overall incentive for invoking microservice), and the device's status (e.g., CPU power, memory, CPU usage status, accuracy of sensors). When querying the device for microservice invocation, `Parameters` can be used to describe how many devices are requested (`-n`), as well as the NFRs(`-h=feature` indicates to select device with the highest value of `feature`,`-l` for the lowest). When executing a microservice, `Parameters` can be used to specify the runtime parameters to be bound to the microservice's execution.

Use Cases *Register/Stop Resources:* An MDC device can register its device capability as available for remote execution, as well as stop such sharing. By leveraging such function, the programmers can decide what capabilities to share, based on the device owner's permission and the device's real-time status. The example program given in Fig. 3.18 shows two procedures: 1) reading the user's permission, and get all available device capabilities for remote execution (line 1-3); 2) specifying that when some computationally intensive applications are running and the CPU load is high, stop sharing the compute capability for remote execution (line 4-5).

Query and Execute Microservice: The functionality demanding devices can query for the most suitable MDC devices to execute a microservice, and request to execute the microservice

```
initialize registry
read user's permission and get available resource
reg.run("reg compute, sensor/Cam -t=1800")
if CPU.usage>50
   reg.run("stop compute")
```

Figure 3.18: MCL Example for Claiming Shared Capability.

on the selected device. The example program given in Fig. 3.19 shows how the motivating example 1 can be implemented in MCL. It also comprises two procedures: 1) query and get three devices for executing microservice “faceReco”, with the highest estimation of the execution speed (line 4); 2) split all photos into three equal shares for the three devices, execute “facoReco” microservices for each photo (line 6).

```
1 initialize registry
2 read images: imgs = readDirectory("...");
3 separate into 3 shares: imgs_0, imgs_1, imgs_2
4 devices = reg.run("query faceReco -l=time -n=3");
5 for (IMAGE img : imgs_0) {
6   devices.get(0).run("execute facoReco -img="+img);}
```

Figure 3.19: MCL Example for Executing Facial Recognition.

3.3.3 Device Selection Mechanism

When processing a microservice request, the local gateway first selects a device most suitable to service the request through the *device selection* procedure. The procedure matches between the requirements of executing a given microservice and the capabilities of the available devices.

Revisiting the facial recognition example: a gateway collects information about the available devices, including their CPU frequencies, memory sizes, and current workloads. Upon receiving a request to recognize a face in an image, the gateway consults the collected information to predict how well each device would satisfy the NFRs of the face recognition microservice (in this case, total execution time). However, predicting how fast a device can execute the facial recognition microservice is non-trivial: not only must the gateway be aware of the device's status, but it must also be able to determine how each aspect of that status would affect the total execution time, which is domain-specific knowledge possessed only by the developers of the face recognition microservice.

In our system design, it is the microservice developers who are expected to provide this domain-specific knowledge alongside the microservice itself. Specifically, microservices include an NFR estimation component. Local gateways download microservice packages from the MSM and execute their NFR estimators to select the most suitable device for the corresponding microservices. Next, we describe the device selection procedure in detail.

Web Interface on Local Gateways The local gateway provides two web interfaces, for MDC devices to register their capabilities, and for microservice demanding devices to query for suitable server devices.

```
Interface 1: resourceRegistry
Parameters: resource = String
                  t = numeric
                  c = String
      {device status = numeric} +
Return: [Registration Success|Fail]
```

Figure 3.20: Capability Registration Interface

Fig. 3.20 demonstrates the interface for registering device capabilities. The `device status` currently includes CPU frequency, remaining energy status, memory usage, network speed, and sensor accuracy.

Fig. 3.21 demonstrates the interface for querying for suitable server devices. The client needs to provide a microservice ID, how many devices to select(`n`), and the NFRs (`h`/`l` for the highest/lowest estimated value).

```
Interface 2: deviceSelection
Parameters: Microservice = String
                        n = numeric
                        h = String
                        l = String
Return: [Connection info of Devices|null]
```

Figure 3.21: Microservice Selection Interface

Estimating NFR Satisfaction Upon receiving a device selection request from a client, the gateway downloads the NFR estimation component of the required microservice from the MSM, and starts matching the device capability and execution requirements. Fig. 3.22 demonstrates an example of the NFR estimation package for microservice `faceReco`. Method `isCapable` checks whether a device is capable of executing a given microservice, and methods `energy` and `time` estimate how a device would satisfy these two NFRs, respectively.

```scala
class FaceRecoEstimator(val d: Device)
                     extends Estimator {

  override def isCapable(): Boolean =
              { d.compute().available() }

  def energy(): Int = 100 - d.battery.toInt

  def time(): Int = {
    var ret: Int = d.CPU * (1 - d.CPUusage)
    if (d.memory > 2000) ret *= 2
    ret
  }}
```

Figure 3.22: Estimating NFR Satisfaction (in Scala)

Revisit the device selection request expressed in MCL script, as shown in Fig.3.19. Upon receiving the request, the gateway first finds a set of nearby devices, whose `isCapable` methods return true. Then, it executes the `time` method on each device, selects the three devices with the lowest expected execution times, and returns the information to the requester about how to connect to these three devices.

3.3.4 Reference Implementation and Evaluation

We report on 1) the reference implementation of the described architecture; 2) the performance of the implementation; 3) the comparison between our device selection procedure and that of key other designs. We implement the local gateway on a off-the-shelf WiFi router, a generally available infrastructure component, thus indicating the wide applicability of our system design.

3.3.5 Implementation Specifics

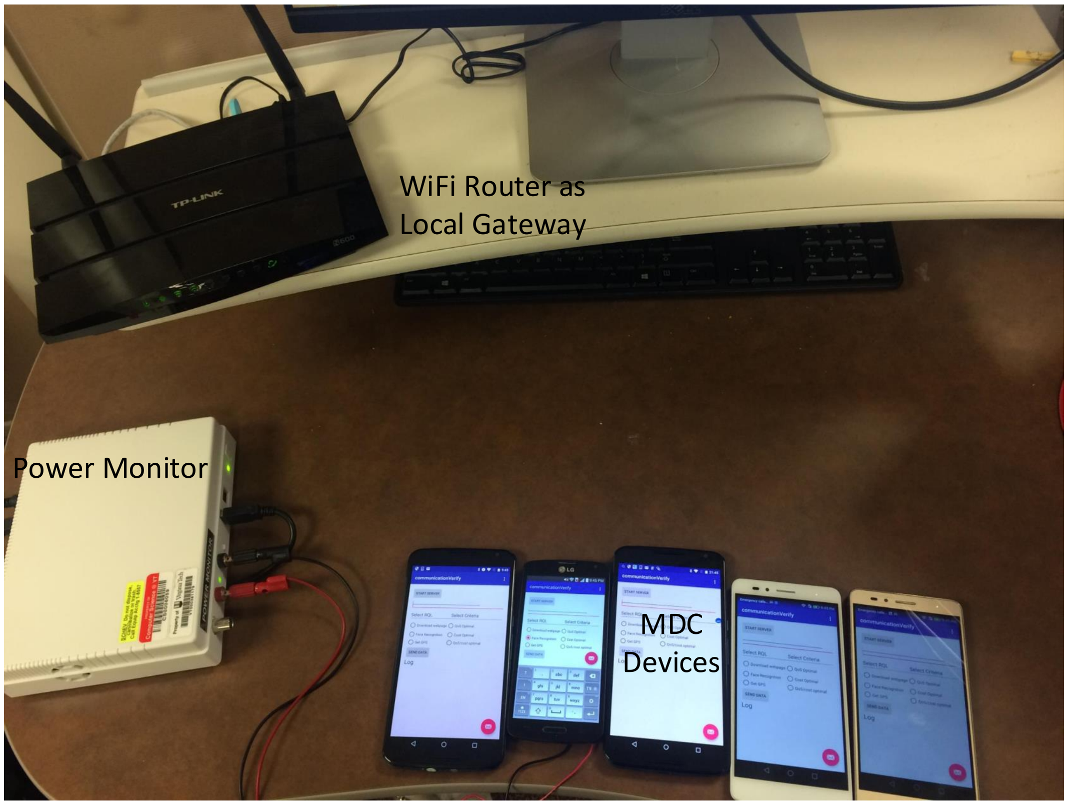
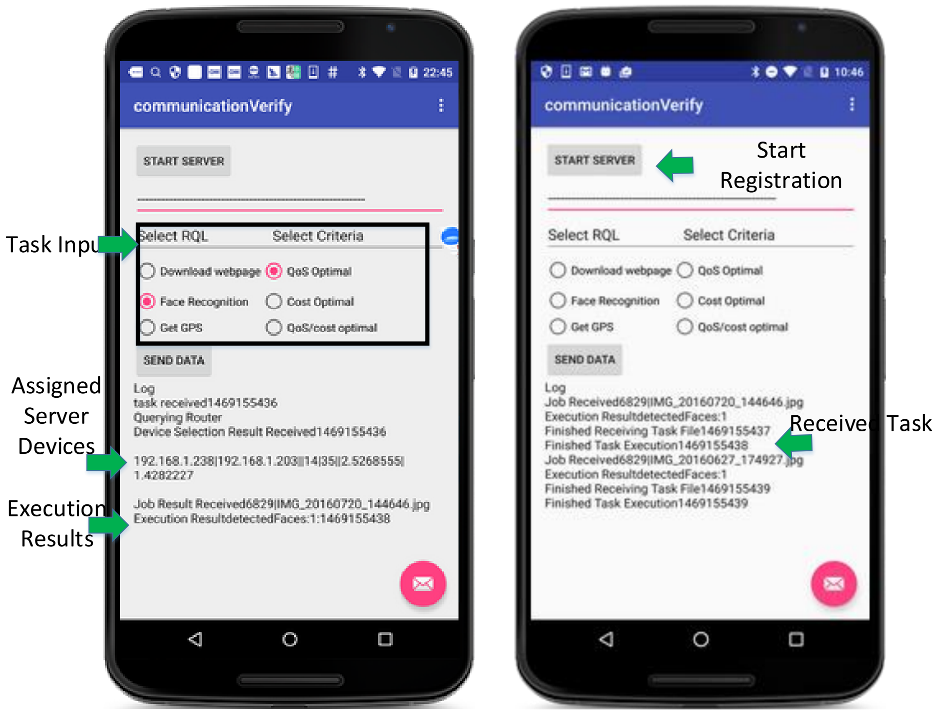


Figure 3.23: Hardware for the Implementation and Evaluation.

Fig.3.23 shows our evaluation's hardware components, which include two Nexus 6 phones, two Huawei Honor 5x, one LG Volt Phone, a Monsoon power monitor, and a TP-LINK TL-WDR3600 router. To make the WDR3600 router serve as the local gateway, we flush openWRT system image to replace the system image provided by the vendor. openWRT system is a Linux distribution for embedded devices. We further install PHP, MySQL and nginx to provide web services, and develop the corresponding PHP script files for the aforementioned interfaces.

For evaluating MCL, we develop a distributed app, whose client and server parts run on microservice invoking devices and the MDC devices, respectively. For MDC devices, their user decides whether to start or stop sharing device capabilities via a simple button click, which sends the corresponding MCL script to the local gateway. For the microservice invoking devices, their users generate different request combinations of microservices and NFRs. We implement and evaluate three microservice packages: file download, face recognition, and get GPS. To simplify the device selection requests, we define the same NFRs for all these three microservices, namely `QoS`, `cost`, and `efficiency` (QoS/cost).

Fig.3.24 shows the runtime procedure of executing the microservice of face detection. The cost of performing face detection is determined by the remaining battery level: a lower battery level leads to a higher cost. The QoS of the service execution is determined by the frequency of the CPU: a higher CPU frequency leads to faster execution, and thus higher QoS. One Nexus 6 serves as the client device, and the other four devices serve as available devices. After receiving the service request, the client device first queries the connected router, and obtains the IP address of the assigned server device. It then connects to the assigned device via a socket and sends the package's and function's names, the input parameters, and the image files to process to the server device. After execution, the results are passed back to the client device.



Download webpage [✓ QoS Optimal]
[✓ Face Recognition] Cost Optimal
Get GPS QoS/cost optimal

```
Log
task received1469155436
Querying Router
Device Selection Result Received1469155436

192.168.1.238|192.168.1.203||14|35||2.5268555|
1.4282227

Job Result Received6829|IMG_20160720_144646.jpg
Execution ResultdetectedFaces:1:1469155438
```

(a) Client Device

Download webpage QoS Optimal
Face Recognition Cost Optimal
Get GPS QoS/cost optimal

```
Log
Job Received6829|IMG_20160720_144646.jpg
Execution ResultdetectedFaces:1
Finished Receiving Task File1469155437
Finished Task Execution1469155438
Job Received6829|IMG_20160627_174927.jpg
Execution ResultdetectedFaces:1
Finished Receiving Task File1469155439
Finished Task Execution1469155439
```

(b) Server Device

Figure 3.24: Execution UI.

Performance Evaluation **Device Selection:** For each microservice, we test different NFRs, to simulate the dissimilar requirements that can be imposed on the device selection criteria (e.g., some may want the service to be executed as fast as possible, while others may want to incur the smallest costs). When the criteria is `QoS optimal`, the Nexus 6 is selected, because it has the highest CPU frequency. When the criteria is `Cost optimal`, the LG Volt is selected, because it is connected to an external power supply.

Execution Time: We repeat the experimental execution 10 times, and calculate the average time taken by each procedure on the client device. We observe that, the time consumption for microservice execution device selection is low (0.15s), compared with the time cost of establishing a connection to the selected device(0.61s), and executing the microservice(1.26s). For the MDC device, the average time consumed to register its capabilities is 0.87s, because

Number of Devices	1	10	20	50	100
Server Device Query (ms)	14	90	171	377	531
Capability Registration (ms)	18	110	192	461	563

Table 3.4: Gateway's Average Response Time.

it needs to obtain the device's real-time status. Although the registration time is close to one second, this latency should not affect the perceived system performance; while the device information is being updated, the old device information can still be used simultaneously.

Energy Consumption: We record the energy consumption of the LG Volt device in the idle state for 30 seconds, and record the energy consumed by querying the microservice execution device/registering device capability once per second for 30 seconds. To protect the result from being distorted by the caching strategy of the Android Volley library, we add a random parameter to each request.

Our experiment shows that, the energy consumption for the client device to parse the MCL request and obtain the assigned MDC device from the WiFi router is 0.009 mAh; the energy consumed by the MDC device to register with the WiFi router is 0.023mAh. If an MDC device registers with the gateway once per minute for one day, the overall energy consumed would be 33mAh, and this energy expense should not affect the experience of mobile users, given that the battery capacity of a typical modern smartphone is at least 2000mAh.

Performance of the Gateway: We use ab to benchmark the performance of the HTTP services, including registering device capability and querying for microservice execution devices, provided by the WiFi router. We run this test on a notebook that connects to the router via WiFi. We simulate 1, 10, 20, 50, 100 devices connecting to the router simultaneously, and Table 3.4 shows the average execution time. As the bulk of the processing load takes place in the WiFi router, the obtained results show high scalability even when stress testing the system with an unrealistic number of requests to the router.

Device Selection Procedure We also experiment with comparing our device selection procedure with that of other state-of-the-art systems. Table 3.5 gives the description of three key competing designs with 3) being our system.

Device Discovery	Energy	Latency	Programmability
BLE Broadcast	Low	1.26s	Low
UDP Broadcast	Middle	0.38s	Low
Router as Gateway	Middle	0.2s	High

Table 3.5: Properties of Device Selection Mechanisms.

1) BLE Broadcast Based [88]: The functionality demanding devices use the BLE broadcast to announce their requirements. When the MDC devices receive the broadcast, they connect to the broadcasting device, and transfer their device capability to it. For the broadcasting device, if multiple MDC devices can provide the required functionality, it needs to wait for all MDC devices to respond, and then select one device that best fits the NFRs, and establish a BLE connection with that device for executing functionality remotely.

2) UDP Broadcast Based [70, 89]: MDC devices are all connected to a local network. The functionality demanding device sends out a UDP broadcast, with the required functionality, the NFRs, and the IP address of the device included in the broadcast message. When an MDC device receives the broadcast and determines that it fits the requirements, it sends its information back to the broadcasting device. The broadcasting device waits for all nearby devices to respond, and then starts a socket connection with the device that best fits the NFRs.

Here we compare the performances and applicability of all the considered device selection strategies:

1. **Energy.** Table 3.6 shows the comparison of the amount of energy consumed by each strategy over time. BLE is the most energy-efficient, while the other two methods consume

slightly more energy.

Execution Time	2h	4h	6h	8h
Stand By	93 %	87 %	79 %	71 %
BTLE D2D Broadcast	93%	86%	78%	70%
Node in WiFi Cluster	92%	84%	76%	68%

Table 3.6: Remaining Battery Percentage Over Time.

2. **Latency.** Table 3.5 shows the latency result of our experimental implementation, with an MDC comprising three devices. We conclude that 1) The latency of BLE is the highest, because all MDC devices need to connect to the resource requesting device, and pass their capacity to the device via BLE communication, which is rather slow. 2) the UDP broadcast strategy also incurs higher latency than the gateway-based ones. We further increase the number of the MDC devices to 5, and observe that the latency of both UDP and BLE broadcasts increase accordingly.

3. **Programmability.** We evaluate the programmability of these strategies, in terms of uncommented lines of code it takes to implement each functionality. When registering device capabilities, our strategy takes 33 ULOC, with the majority of the code written to obtain the device's status. The two broadcast based strategies take 57 and 86 ULOC, respectively, due to them needing to manage the D2D communication. When selecting devices, our strategy takes only 5 lines of code, with the broadcast based strategies taking over 200 ULOC.

Based on this evaluation, one can conclude that our gateway-based system architecture enables mobile apps to leverage MDCs with low latency and high energy efficiency. In

ULOC	Register & Stop	Device Selection
Router-based	33	5
BLE Broadcast	86	231
UDP Broadcast	57	208

Table 3.7: ULOC for Each Function.

addition, our architecture's device selection procedure requires fewer lines of programmer-written code as compared to the broadcast-based alternatives.

3.4 Related Work

Using nearby mobile devices to cooperatively implement new functionality was originally proposed as a means of exchanging private information over devices for data sharing and data mining [81]. Subsequent research took user mobility into account [64, 83, 126].

Besides data sharing, another avenue for device cooperation is running map reduce [183] on mobile devices to execute computational-intensive tasks [36, 112, 147]. These approaches, however, are oblivious to device mobility and the preference of users to participate.

In addition to traditional mobile devices, the IoT setups can provide resources for device-to-device resource sharing. Computational tasks have been offloaded to such setups (e.g., Road Side Unit) [52], while mobile messages have been stored and forwarded by a wall-mounted Estimote device [11]. The proposed project will focus on the software engineering aspects of mobile device cooperation, thus benefiting the implementation practices of many of the prior state-of-the-art approaches.

Traditional middleware has been adapted for peer-to-peer resource sharing, including Open CORBA[106], Globe[166] and JXTA[48], although without taking device mobility into account. The UPnP protocol[150] enables network devices to provide service to other devices in the network.

Device mobility-aware peer-to-peer resource sharing has started from content sharing [127], with numerous subsequent approaches [7, 19, 39, 69, 78, 120, 122, 140]. Special purpose middleware support face-to-face interactions [142] and cooperative display[12]. These middleware approaches are platform-specific and require modifications at the system level. By contrast, the proposed project aims at heterogeneous device-to-device applications running on top of unmodified system stacks.

The MANET project leverages assistance from devices through multi-hop wireless communication [33]. Various middleware approaches have focused on various aspects of inter-device cooperation, including LIME[119], TOTA[110], Limone[44], CAST[136], MESHmdl[56], Preom [80], MobiPeer [16], Peer2Me [173], Steam [115], Transhumance [128], QAM [45], and MobiCross [34]. These middleware approaches provide programming to control network topologies, network traffic, peer management, etc. By contrast, the proposed approach focuses on supporting mobile application programmers, who are primarily concerned with obtaining the hardware resources they need for their applications.

To support platform independence, [129] proposed using an HTTP server. By contrast, this project focuses on P2P communication, thus reducing communication latencies and processing overhead.

3.5 Conclusion

This chapter is concerned with the problem of how to improve the performance, energy consumption, and latency of mobile applications by sharing resources across nearby mobile devices. Although many prior research publications have focused on cooperatively sharing resources across devices to enable new functionalities or to optimize energy consumption and runtime performance, application developers lack software engineering support for seamlessly sharing resources between heterogeneous mobile devices. To address this problem, we first study how to encapsulate remote execution on collocated devices as mobile microservices, supported by a novel distributed system component. Second, we explore a D2D approach and an edge-gateway based approach for executing services on collocated devices. We also empower developers in creating applications that integrate such remote executions in mobile applications by providing intuitive programming models. We carefully compare the performance characteristics of these two approaches, and conclude that the edge-gateway based approach is more efficient in terms of latency and energy consumption. Hence, the rest of this dissertation research is built upon the edge gateway system architecture. By facilitating the process of implementing cooperative resource sharing among devices, our ultimate objective is to add this support to the standard toolset for mobile application developers.

Chapter 4

Microservice Orchestration Language with Support for Equivalence

In contrast to cloud computing, edge computing processes data locally near its source (i.e., at the "edge" of the network), thereby reducing the network transmission load and communication latency. In addition, by leveraging the edge environment's sensor and networking resources, edge computing applications can take advantage of the local context and accelerate data transfer [6, 13, 27, 28, 73, 96, 102, 156].

The need to access nearby sensors and to reduce communication latencies requires that edge resources be orchestrated for a reliable and efficient execution. Nevertheless, software developers lack adequate programming support to be able to engineer such edge computing applications [167, 168, 187]. In recent years, microservices [116] have been embraced as an architecture that structures distributed systems modularly to clearly separate concerns. Microservices fit naturally the domain of edge computing, which coordinates the execution of multiple dissimilar computing devices. However, extant microservice frameworks are inherently cloud-based, and cannot be directly applied to edge-based environments.

Two primary factors hinder the use of microservices at the edge: (1) Cloud-based microservice architectures require that all executable resources be pre-deployed on the participating devices, which can be accessed by querying an Internet-based registry service. However, connected via local-area networks, edge-based resources can only be accessed within a limited physical area. (2) As a result, edge environments differ in their setups, making it impossible to rely on any standard set of edge-based resources. Hence, robust and efficient edge computing applications should be able to adapt to the available sets of resources in dissimilar runtime environments.

Consider obtaining environmental sensor data, such as temperature, humidity, or CO_2. A mobile application may need to keep track of up-to-date environmental data, specific to the device's current geo location. However, edge environments often possess dissimilar resources that can provide the necessary data. For instance, temperature can be read from a local sensor or be obtained by passing the location parameter to a web-based weather service. To fulfill these application requirements, developers need to either implement complex logic that covers all possible combinations of available edge resources, or hardcode the implementation for a particular edge environment with pre-deployed resources.

In this chapter, we present a novel programming model for orchestrating reliable and efficient execution in edge environments with variable resources. Our model features a declarative domain-specific language (DSL) for orchestrating the execution of microservices at the edge. Our language is called MOLE (**M**icroservice **O**rchestration **L**anguag**E**). The MOLE compiler takes as input the declarative specification of microservices and produces a platform-independent execution plan. The MOLE runtime takes the generated execution plan as input, and adaptively steers the execution of the expressed functionality on the set of available devices.

The contribution of this chapter is four-fold:

1. We present MOLE—a declarative DSL that enables programmers to express edge-based application as an ensemble of microservice executions; MOLE naturally supports redundant execution to adapt to opportunistically available resources.

2. We describe the MOLE compiler that generates platform-independent execution plans; the compiler automatically parallelizes microservice execution.

3. We design a novel microservice-based runtime architecture that supports MOLE programs to execute microservices on the available edge devices.

4. We evaluate MOLE, its compiler, and runtime system on a set of benchmarks and case studies.

4.1 Problem Analysis

In this section, we demonstrate the difficulties of programming edge computing applications.

4.1.1 High Resource Variability at the Edge

Developing software for edge computing environments differs from that for the cloud. Developers can reasonably assume the high availability and reliability of cloud-based resources. Cloud providers are bound by the terms of Service Level Agreements (SLAs) to ensure their services remain up and running. Hence, because most failures in cloud-based systems are recovered from quickly, a simple retry to contact a temporarily inaccessible cloud service is a reasonable fault handling strategy [62]. However, in edge-based environments, the resource availability is likely to cause execution failures, triggered by the differences in the resource setups of edge environments.

Nevertheless, edge programming models [62, 130] continue to follow the fault handling strategies, originally introduced for cloud-based microservices — handling faults by retries and adjusting minor configuration setups (e.g., switching network connectivity methods, switching to devices capable of providing the same functionalities).

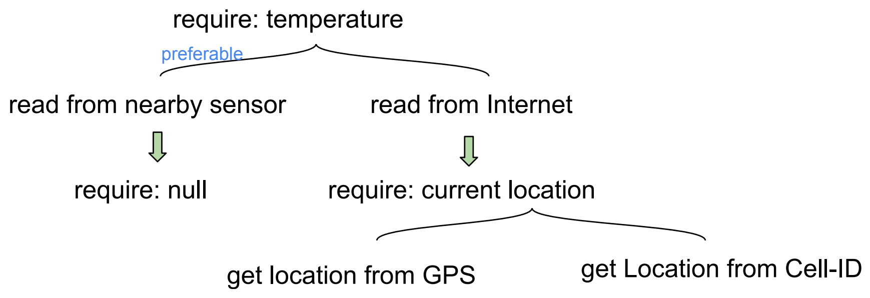


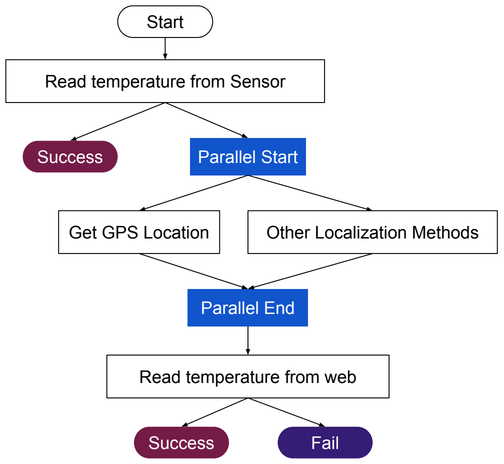
Figure 4.1: Increasing Dependability by Increasing Redundancy.

We observe that edge environments can provide the same application functionality in a variety of ways. In the motivating example, the developer can either read a local temperature sensor or parameterize a web-based weather service with the user's geo location. As another example, consider detecting the breakout of fire in a building, different sensors (e.g., temperature, dust level, CO_2 level, etc.) can be combined to ascertain whether there is fire.

Hence, given the high variability at the edge, our programming model centers around the concept of *resource redundancy and makes it natural for the developer to specify alternative ways to provide the same functionality.* Continuing with the temperature example above, Fig. 4.1 shows a possible design flow, in which the developer first considers obtaining temperature by reading a local sensor, but then realizing that such sensors may be unavailable or disabled, would specify a back-up alternative of obtaining the required information from a web-service. Both alternatives provide equivalent functionalities with minor caveats. Local sensors are likely to provide higher accuracy, while weather web services are highly reliable, even when given a coarse-grained geo location. To obtain the location, multiple localization methods (e.g., GPS based, cell-id based, WiFi based) are equally suitable.

4.1.2 Complexity of Orchestrating Edge Microservices

To implement the redundancies, developers typically need to engineer high-complexity code, particularly if the resulting execution has efficiency requirements.


Start
Read temperature from Sensor
Success
Parallel Start
Get GPS Location
Other Localization Methods
Parallel End
Read temperature from web
Success
Fail
Figure 4.2: Execution Sequence of Example Edge Application.

Let us revisit the temperature example above. Some localization methods can experience unexpectedly high latencies. To accelerate the overall execution, the developer may want to take advantage of speculative parallelism: spawn multiple localization methods at once, and proceed once any one of them returns successfully. Fig. 4.2 shows how speculative parallelism can be integrated into the execution flow.

There is an impedance mismatch between the simplicity of how developers can divide a required functionality into distinct functions (Fig. 4.1), and the complexity of orchestrating these functions to execute correctly and efficiently (Fig. 4.2). Existing programming models require that functions be explicitly arranged into an execution flow, thus unnecessarily burdening the developers.

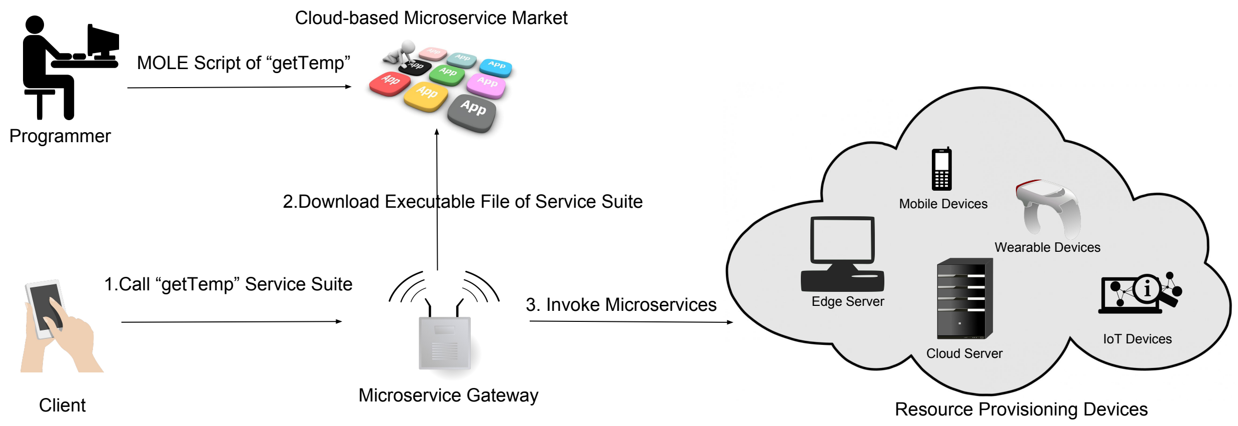


Figure 4.3: Solution Overview.

4.2 MOLE Overview

Next, we give a brief overview of how developers can use MOLE to provision for edge applications. First we briefly introduce the system architecture, and then explain the basic system execution flow of MOLE applications.

4.2.1 System Architecture

Fig. 4.3 shows the MOLE system architecture, which comprises four major components: 1) a client device that requires distributed resources to accomplish an application functionality; 2) a local device that serves as a gateway by maintaining an up-to-date mapping between the available nearby devices and their resource capacities; 3) a microservice market, a cloud-based repository of executable code of all available microservices; 4) a set of local devices that provide their resources to applications.

Gateways: A typical microservice architecture features a centralized service registry, a collection of registered microservice-to-device mappings, with a remote interface through which clients can bind themselves to the microservices they want to invoke. Notice that if the registry is not replicated, it becomes vulnerable to the single point of failure. Besides, edge-based applications need to invoke microservices on the devices reachable via short-range communication methods (e.g., WiFi, Bluetooth), rendering cloud-based registries inapplicable. To meet these requirements, MOLE features a novel system component: an edge gateway, thus replacing the standard cloud-based service registry. Similarly to its cloud-based counterparts, the edge gateway maintains a registry of all the microservices provided by edge devices. At runtime, clients interact with the reachable gateway in its vicinity to execute microservices; the gateway interacts with the available devices on its clients' behalf. Since gateways form a network, in case a device hosting a gateway fails, clients proceed contacting the remaining gateways until reaching one of them.

Cloud-based Microservice Market: Our design leverages the **M**icro**S**ervice **M**arket (MSM for short), a cloud-based network component that combines features of application markets and service repositories 3.1. By following the application market model, MSMs enable devices to automatically download the needed microservices for execution. By following the service repositories model, MSMs enable edge application developers to implement the needed functionalities as microservices, to be executed by the available devices in a given edge computing environment.

4.2.2 Service Suite Execution Model

To understand the general MOLE system flow, recall the "getting the temperature" example. An application running on a client device sends the request to execute `getTemp` service suite to a nearby gateway (Step 1). The gateway downloads the `getTemp` service suite from a cloud-based MSM (Step 2), and executes it by orchestrating the microservice invocations on the available devices at the edge (Step 3). The gateway continuously collects the microservice execution results, which drive the orchestration of the microservice invocations involved.

Upon completing the service's execution, the gateway returns the final results to the client.

In the example above, `getTemp` comprises a collection of microservice invocations, which can be initiated by edge applications to obtain the functionality at hand. In the rest of the manuscript, we refer to such collections as **a service suite**.

Definition 1. Service Suite: implements an application functionality by orchestrating a collection of microservices.

The MOLE programming model enables service suite developers to declaratively specify how to orchestrate the execution of microservices. The MOLE compiler then translates these specifications into an execution graph, while optimizing the resulting edge based execution via speculative parallelism. The MOLE distributed runtime finally discovers the available devices to execute the specified microservices on them, as directed by the compiled MOLE specifications.

4.3 MOLE DSL Design

Fig. 4.4 defines the syntax of MOLE in EBNF. Some of the key features are as follows:

- Each service suite is identified by a unique id, `Service Identity`. Service suites may take `Service Parameter`, which must be passed when the suite is invoked.
- A service suite comprises one or more `Microservice Invocation`'s, identified by unique IDs, and containing additional attributes explained next.
- A microservice invocation comprises the following attributes: 1) the `Device Selection` rules that guide how to select a device to run on; 2) the `Input Params` that specify the microservice's invocation parameters, some of which are hardcoded (indicated by

```
<Service_Suite> ::= <Service_Identity> <Service_Description>
<Service_Identity> ::= "Service "String
<Service_Description> ::= "{"[<Service_Parameter>] <Microservice_Invocations>"}"
<Service_Parameter> ::= "global_input: "[<Input_Parameter_Name> ","]
<Microservice_Invocations> ::= [Microservice_Invocation]+

<Microservice_Invocation> ::= "MS:" <MS_Identity> "{" [<MS_Detail>]+ "}"
<MS_Identity> ::= String

<MS_Detail>::= <Device_Selection>|<Input_Params>|<After_Execution_Rules>
<Device_Selection> ::= "device:" [<Select_Rule> "."]+
<Select_Rule> ::= "select"|"sort" "(" String ")"
<Input_Params> ::= ("req": [<Param Name> ","]+)|("set:" [<Param_Name> "to" <Param_Value>
     ","]+)
<After_Execution_Rules> := "on." <Condition> ":" [ <return> ";"] [<redirection>]
  <Condition> ::= "success"|"fail"|"res."<Param_Name><Operation><Value> |"ep."<Param_Name
      ><Operation><Value>
  <return> ::= "ret" [String ["as" String] "," ]+
  <redirection> :== <MS_Identity>|"exit"
```

Figure 4.4: DSL EBNF Definition.

```
Service getTemp {
    MS: getTempSensorReading {
        device: select("Sensor.Temperature")
        on.success: ret temp
        on.fail: getTempbyLocation
    }
    MS: getTempByLocation {
        device: select("Internet")
        req: location
        set: ep.max_retry to 3
        on.success: ret temp
    }
    MS: getLocationByGPS {
        device: select("Location.GPS_PROVIDER")
        on.success: ret loc as location
    }
    MS: getLocationByCellID {
        device: select("Location.NETWORK_PROVIDER")
        on.success: ret loc as location
    }
}
```

Figure 4.5: Source File of `getTemp` Service Suite

`set`) while others are passed at runtime (indicated by `req` – short of "require"); 3) the `After Execution Rules` that specify what results should be returned (`ret`), and what the next suite execution step should be (could be either `exit`ing service suite , or invoke another `MS`).

- The execution procedure of a microservice can be controlled by execution parameters (`ep`), which is a special kind of `Input Params`. `ep` contains a fixed set of directives: `maxExecutionTime`, `maxRetry`, `retryOnOtherDevices`, and `counter`.

As a concrete example, consider the MOLE script in Fig. 4.5, which describes the `getTemp` service suite. `Service Identity` is `getTemp`, a service suite that takes no parameters. It comprises four `Microservice Invocation`: `getTempSensorReading` (m_1), `getTempbyLocation` (m_2), `getLocationbyGPS` (m_3), and `getLocationbyCellID` (m_4). Each microservice has `Device Selection` rules and `After Execution Rules`, while only m_2 needs `Input`.

A pair of microservices (m_1, m_2) can relate to each other in two ways: 1) *forward relationship*: m_1 invokes m_2 based on the suite's business logic; 2) *backward relationship*: m_1 has an input parameter, whose value must first be computed by invoking m_2. In a given suite, developers orchestrate the execution of microservices based on the concepts of forward and backward relationships.

To provide an intuitive programming model, MOLE requires that **only the forward relationship be explicitly defined** (e.g.,invoke m_2 iff m_1 fails). Backward relationships are automatically inferred based on the naming correspondences between the input and output parameters of the microservices in a suite (e.g., m_2 requires input parameter 'a', m_3 produces 'a' as its execution result, so the compiler orchestrates the correct execution sequence of $\{m_1$, m_3, $m_2\}$).

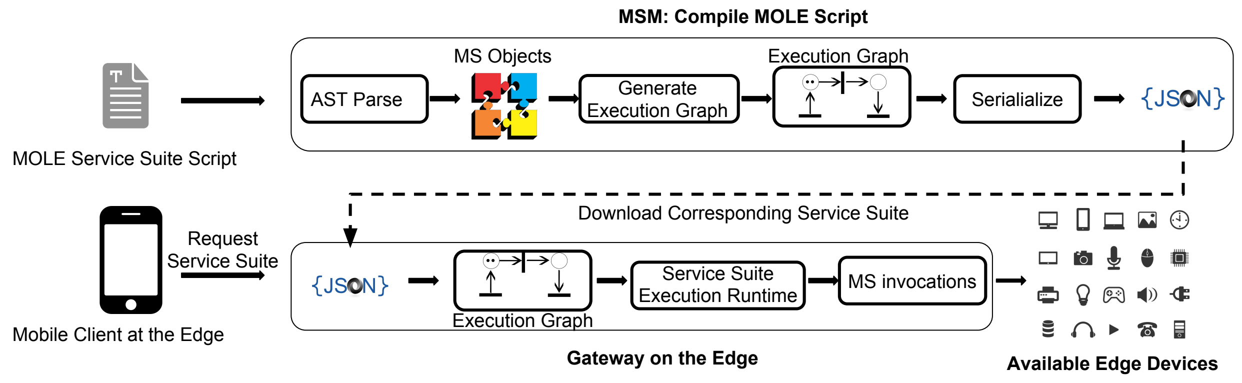


Figure 4.6: DSL Parsing and Execution.

4.4 MOLE Compiler and Runtime

Fig. 4.6 shows how a MOLE script file is compiled, optimized, and executed. Upon completing a microservice suite, developers upload them to the mobile service market (MSM) containing the referenced microservices. Recall that MSMs are network components that combine features of service repositories and app markets. An MSM has the facilities for error checking, compiling, and optimizing MOLE suite specifications. The end result of processing a specification is an executable containing the service suite's **Execution Graph**, a self-contained repository for all the information required to efficiently execute the suite. Once a client invokes the edge application, the edge's gateway component downloads the compiled Execution Graph from the MSM and starts executing it. The execution also involves downloading the referenced microservices to the devices selected to execute them. If the gateway fails for any reason, the edge app's client can always start interacting with an alternate gateway component, thus providing a fail-over fault handling strategy. Upon successfully completing its execution, the suite returns the results back to the client, or an error if the execution failed for any reasons.

We first describe execution graphs, and then explain how MOLE scripts are compiled into execution graphs. Finally, we discuss how the MOLE distributed runtime executes execution graphs using a distributed microservice gateway.

4.4.1 Execution Graph Definition

An execution graph $G = (N, E, P)$ comprises a set $N = \{n|n = (t, m, d, p)\}$ of nodes, a set $E = \{e|e = (n_s, n_t, c, a)\}$ of edges, and a set of global parameters P that must be bound before an execution can start.

A **node** $n = (t, m, d, p)$ comprises the type t of the node, the microservice m related to the node, the device selection rules d of the node, and a set of required execution parameters p. An **edge** $e = (n_s, n_t, c, a)$, also written as:

$$e = n_s \xrightarrow[\text{a}]{\text{c}} n_t$$

indicates that when the execution results of a microservice node s fits a condition c ($c = null$ if the type of n_s is not a microservice), the next microservice node to visit is n_t, with a set of arguments a passed to it. n_s is called the *source* of e, n_t is called the *target* of e, e is an *outgoing* edge of n_s, and e is an *incoming* edge of n_t.

The possible **type** of nodes $t \in [M, E, ES, EF, PS, PE]$, where M stands for a microservice node, E for the entry node of the service suite, ES for the successful exit node, EF for the failure exit node, PS for a parallel start node, PE for a parallel end node. There can be only one entry node (n_E), one successful exit node (n_{ES}) and one failure exit node (n_{EF}).

The PS and PE nodes designate the start and the end of a **speculative parallel execution** block, respectively. Upon entering a PS node, all parallel branches start executing their first nodes (linked by the outgoing edge of the PS node) in parallel. A parallel branch may have multiple MS nodes to execute, and all parallel branches aggregate at the peer PE node. When all required parameters of the PE node have been provided by any combination of branches, the PE node starts to execute its next MS node, disregarding the completion statuses of the remaining parallel branches.

4.4.2 Generating Execution Graphs

The MOLE compiler transforms an input script file into an execution graph. The key idea of the transformation is to run a two-phase analysis: (1) control-flow analysis adds edges between pairs of microservices on the control path (e.g., if n_s fails, invoke n_t) and (2) data-flow analysis adds edges between microservices with data dependencies, (e.g., n_t takes as input the n_s's execution result). The required parallel blocks are added into the execution graph during the second phase.

Algorithm 1 controls the transformation in four basic steps:

(1) Initialize Nodes (Line 3 - 5): convert each microservice declared in the source script into "MS" nodes. The node structures encapsulate the microservice invocations, device selection rules, and required input parameters. Each graph also includes four special nodes: N_E (entry node), N_{ES} (execution success) , N_{EF} (execution failure), and $Null$ Node N_N, a temporary placeholder used at graph construction time.

(2) Initialize Edges via Control Flow Analysis (Line 6 - 17): parse the MOLE script to extract the conditional statement for each "MS" node. Recall that only the forward relationships must be defined explicitly. If a node's conditional statement is linked to another node (could be either "MS", N_{ES}, or N_{EF} node, which define the forward relationships), add an edge to the execution graph connecting the two nodes; otherwise, if it only generates data as output (e.g., 'on.success: ret loc as location", which can be used to infer the backward relationships), add an edge to the graph, connecting the node with the special N_N node.

(3) Add Edges via Data Flow Analysis (Line 18 - 21): generate a set, initialized with the "MS" nodes, except those connected to the N_N node. All edges leading to the N_N node

Algorithm 1 Generate Execution Graph.

```
 1: function generateExecutionGraph()
 2:     ExecutionGraph eg ← ExecutionGraph()
 3:     // Step 1: init nodes
 4:     eg.nodes ← ParseMicroserviceNodes()
 5:     eg.nodes ← eg.nodes + $N_E$, $N_{ES}$, $N_{EF}$  $N_N$
 6:     // Step 2: init edges by control flow analysis
 7:     for all node ∈ eg.nodes do
 8:         for all c ∈ node.conditions do
 9:             if c.type == "invoke microservice" then
10:                 e ← Edge(node, c.target, c.condition, c.params)
11:                 eg.edges ← eg.edges + e
12:             else
13:                 e ← Edge(node, $N_N$, c.condition, c.params)
14:                 eg.edges ← eg.edges + e                          ▷ link to node Null
15:             end if
16:         end for
17:     end for
18:     // Step 3: add edges by data flow analysis
19:     dataEdges ← $N_N$.getIncomingEdges()                            ▷ GIE() for short
20:     loopNodes ← eg.nodes - specialNodes - dataEdges.s
21:     loopNodes.BFS()                          ▷ Breadth-First-Search for adding edges
22:     // Step 4: find entry node
23:     entryNodeSet ← EmptySet
24:     for all n ∈ eg.nodes do
25:         if  n.type=="ms" AND n ∉ dataEdges.s AND n.GIE()=null then
26:             entryNodeSet ← entryNodeSet + n
27:         end if
28:     end for
29:     if  entrySet.size!=1 then
30:         Raise CompileError("Cannot Find Entry Node")
31:     else
32:         eg.edges ← eg.edges + ($N_E$, entrySet[0], null, null)
33:     end if
34:     return eg
35: end function
```

become "dataEdges" to provide missing arguments for other microservices. For each node's incoming edges, calculate whether the incoming edge's bound arguments can serve as the microservice's required parameters. If not, check if the missing parameters can be provided by "dataEdges". If only one "dataEdge" can provide the missing parameter(s), add the source of the "dataEdge" to the graph, between the current node's source node and the current node. If more than one "dataEdge" can provide the missing parameter(s), add a pair PS, PE of parallel blocks between the source node of the current node and the current node, and then add all “dataEdges” into the parallel blocks. When new edges are added, the edges’ source nodes are added to the set, so they can be also properly processed. This step is actually applying the breadth-first search algorithm.

(4) Find Entry Node (Line 22 - 34): for all “MS” nodes, find those without any incoming edges or connected by “dataEdges”. If only one such node is found, add an edge between the entry node and the found node. Otherwise, throw a compile error.

4.4.3 MOLE Runtime

MOLE features a distributed runtime system that efficiently and reliably executes compiled scripts. The runtime’s pivotal component is an *edge gateway*, responsible for collecting the real-time status of surrounding edge devices, accepting service suite execution requests, downloading the corresponding compiled MOLE scripts from MSMs, and invoking the constituent microservices. Each microservice-executing device runs a light-weight HTTP server, which dispatches the referenced microservices by invoking their execution packages, provided on demand by MSMs.

To execute a compiled MOLE script, the edge gateway’s runtime starts the execution at the entry node, moving through the connected nodes to the end node. The execution goes from node to node as follows. When visiting a microservice node, the runtime invokes the microservice, and determines what the next node should be based on the invocation results. For a parallel start node, the runtime spawns concurrent branches, with each branch proceeding along its own path and finally aggregating at the following parallel end node. For a parallel end node, the runtime waits until either the concurrent branches provide the required parameters, or all of them experience faults or timeouts.

4.5 Evaluation

In this section, we evaluate the MOLE programming model and performance in a realistic use cases. Our evaluation seeks answers to the following questions:

- Can MOLE programs adapt to resource variability?
- Does MOLE offer acceptable execution efficiency?
- How hard is it to develop a MOLE program?

4.5.1 Setups

The following discussion first describes the experimental setup, then introduces the executed service suites, and finally reports on the performance characteristics.

The evaluation hardware setup comprises: 1) a wireless router, running the OpenWrt OS; 2) a Chromebook; 3) two Android smartphones; 4) a Raspberry PI; 5) a Dell desktop serving as the edge server, and 6) an AWS cloud-based server (not shown in the Figure). Devices 2)-5) are connected to the wireless router, thus forming a wireless local area network.

A DS18B20 temperature sensor is connected to the Raspberry Pi via general purpose input/output (GPIO). The Raspberry PI hosts a web server that handles POST requests by invoking the corresponding microservice executables. "getSensoryTemperature" is pre-deployed on this device.

The NanoHttpd servers on Android devices invoke the corresponding microservices via reflection in response to incoming HTTP POST requests. One of these devices is configured to provide a fine-grained location, while the other one a coarse-grained one.

The Dell desktop plays two roles: the edge gateway and the edge server. It runs an HTTP server, and a MySQL database. Each edge device communicates with the edge gateway via HTTP to register their microservices; the gateway then persists this information in its database. As an edge server, it runs microservices, such as querying a web service to get the temperature in a given location.

4.5.2 Service Suite Execution

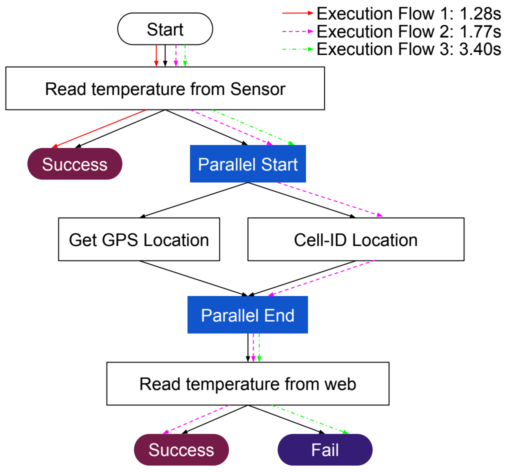


Figure 4.7: Execution Time of Different Availability of Microservices.

End users access a dynamic web page from the Chromebook, which contains a JavaScript function that retrieves the service suite's name, connects to the local gateway (by querying the wireless router), sends the service suite execution request to the gateway, and blocks until receiving the results.

To evaluate how MOLE programs adapt to resource variability, we run our experiments in three dissimilar execution environments. The execution results fit the generated execution graph, as shown in Fig. 4.7:

A Make the Raspberry PI and the temperature sensor available. In this execution environment, the overall execution result obtained by the Chromebook is the temperature measured by the temperature sensor. The average execution time of 5 runs is 1.28 seconds.

B Shut down the Raspberry PI, to make the temperature sensor unavailable. In addition, enable the fine/coarse-grained localizations for the two Android devices. In this run, the overall execution result is the temperature of a geo-area, which differs from the first result. The overall execution takes 1.77 seconds. Please note that in our implementation, the GPS localization takes 2 seconds and the cell-network localization takes less than 1 seconds. This result indicates that even though two localization methods are all initialized, the execution continues when the cell-network location is returned.

C Disable both the temperature sensor and two localization methods. In this run, the parallel start node initializes two threads for two localization methods, but none of them goes to the parallel end node. The timeout for the latch count down is set to 3 seconds, with the parallel end node being reached after the timeout. The parallel end node fails to collect all necessary parameters for its connected node (`getTempByLocation`), so it triggers the execution fail condition of its connected node, thus causing the "Execution Failure" of the service suite. The average execution time of 5 runs is 3.4 seconds.

4.5.3 Programming Effort

Fig. 4.5 lists the source code of the `getTemp` service suite. It takes only 21 lines of code to specify the parameterization of and control-flow between 4 realistic microservices. Under any programming model, programmers have to implement the application functionalities, but representing them as microservices eases reuse. Each microservice is likely usable in multiple scenarios.

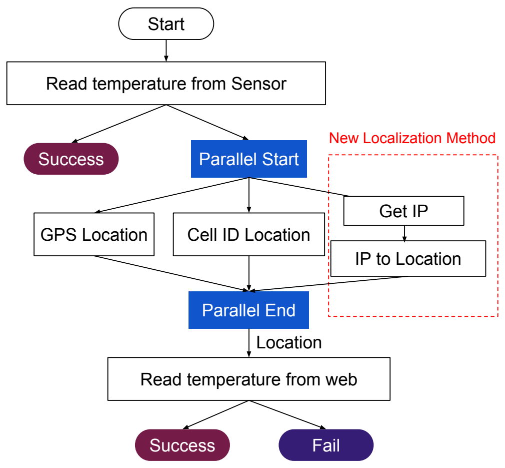


Figure 4.8: New Execution Graph.

A particular strength of the MOLE programming model is how it accommodates change. Consider adding an alternate localization method for MS `getTemperatureByLocation`. Unlike the current two localization methods, the new method operates in two steps: 1) obtain the current IP address; 2) get the location from the IP address using a web service. This change requires only 6 additional lines of code.

```
Service GetTemp {
  ...

    MS: getIP{
        device:has("INTERNET")
        on.success:ret ip;
    }

    MS: IP2Location{
        req:ip
        on.success:ret location;
    }
}
```

Figure 4.9: Adding a new localization method to the service suite

4.5.4 Reliability Evaluation

To assess how reliable MOLE programs are, we simulate the execution of a suite under different failure conditions. This simulation varies the failure rate of each microservice execution between 10%, 20%, 30%, and 90%. Two types of failures apply: (1) no device is available to execute a given microservice; (2) the selected device fails to successfully execute a given microservice. We compare the resulting reliability levels of three scenarios: 1) obtain temperature from a temperature sensor; 2) execute service suite `GetTemp` with two localization methods; 3) execute this service suite with one additional IP-based localization method. Fig. 4.10 shows that compared with scenario 1, the service suite improves its reliability, especially when the failure rate is around 50% (it improves the reliability of scenario 1 by 37.5%). Besides, by comparing scenarios 2 and 3, we see how introducing an alternative localization method increases the overall reliability of the service suite execution. When two existing localization methods fail, the suite can still successfully complete its execution. However, the increase may not seem as striking, as the two alternative localization methods already exhibit considerable reliability.

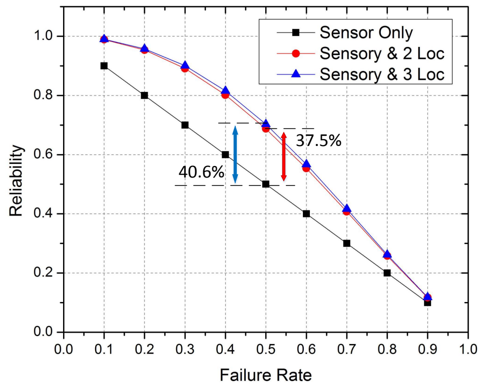


Figure 4.10: Reliability W/ W/O MOLE.

4.5.5 Efficiency Evaluation

Next, we measure how efficient a MOLE program is. We set the execution failure rate of each microservice between 10%, 20%, 30%, and 90%. We compare the total execution time of three scenarios: 1) sequential execution, which runs one microservice at a time; 2) the `GetTemp` service suite executing its two localization methods in parallel; 3) the improved

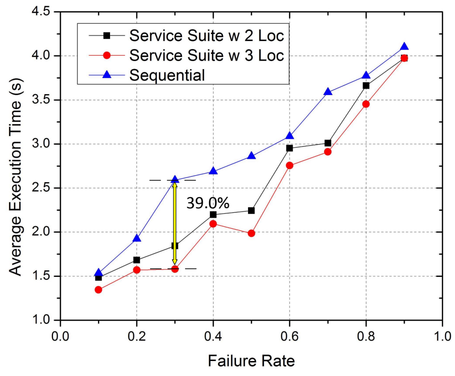


Figure 4.11: Efficiency W/ W/O MOLE.

`GetTemp` service suite executing three localization methods in parallel. For each failure level, we repeat each execution scenario 100 times, and record the average execution time. Fig. 4.11 shows that MOLE can increase the base line of the sequential execution by 12-17% by leveraging speculative parallelism. At most, MOLE saves 39% of execution time when the failure rate is 30%. Besides, because IP-based localization is known to be more efficient than other location methods, MS `getTemperatureByLocation` can proceed without waiting for the other two slower localization methods to complete, thus improving the overall efficiency.

4.6 Related Work

Recent survey papers treat the issue of programming edge applications as both a serious technical challenge and a research opportunity [167, 168]. [59] introduces a P2P message exchange based programming model, by which programmers develop functionalities for each distributed component and handle their communication. However, such programming models can only be applied to execution environments with fixed resources. [149] considers the resource dynamicity of edge computing environments, and models edge service provision as a QoS-constrained resource selection problem. [47] provides a data-flow based programming model, also applicable to edge environments with dynamic resources. However, these approaches neglect failure handling, an essential provision given the high failure ratio of edge-based execution.

4.7 Conclusion

This chapter presents MOLE, a declarative DSL for developing reliable and efficient edge computing applications. MOLE adopts the microservice architecture, with edge functionalities provided as microservices, downloaded and executed by available devices at runtime. MOLE enables developers to concisely express how to parameterize microservices, and automatically orchestrates their execution flow. MOLE exploits the presence of equivalent microservices to orchestrate both fail-over and speculatively parallel execution workflows. Our evaluation has demonstrated the expressiveness, reliability, and efficiency of the MOLE programming model.

Chapter 5

Workflow Support for Equivalent Functionalities

The previous chapters demonstrate that the applicability of the microservice architecture has extended beyond traditional web services, making steady inroads into the domains of IoT and edge computing. A QoS-optimal service balances reliability, execution costs, and latency to satisfy application requirements. In emerging distributed environments, with their unreliable and resource-scarce mobile/IoT devices, it is hard but essential to optimize the QoS of mobile services. Fortunately, these environments are characterized by ever-growing equivalent functionalities that satisfy the same requirements by different means. The combined execution of equivalent microservices has been used to improve QoS (e.g., majority voting for accuracy, speculative parallelism for latency, and failover for reliability). These executions are commonly described as workflow patterns, crude-grained recurring interactions across microservices within a service. However, the current workflow patterns provide limited support for equivalent microservices, causing services with equivalent microservices to suffer from unsatisfied and severely unbalanced QoS. In this chapter, we present two works that introduce dedicated workflow support for equivalent microservices, one automatically enhances service execution efficiency and reliability, and the other automatically generates fine-grained workflow patterns for QoS-optimal combined execution of equivalent microservices.

5.1 Equivalence-Enhanced Microservice Workflow

Service-oriented software development has embraced the microservices architecture [4], dividing a complex software system into coherent and lightweight microservices, each of which performing a cohesive business function. Although traditionally the microservice architecture is used mainly for composing web services/applications[90], emerging application domains, including IoT and edge computing, have started to increasingly apply this architecture as well[144, 171].

If different microservices fulfill the same application requirement, these microservices provide *equivalent* functionalities that can be used in place of each other. Known application patterns that use equivalence include improving reliability via fail-over and reducing latency via speculative parallelism. In the realm of web applications, service equivalence has been applied to select services: choose the one with the optimal QoS features from its equivalent set [158]. Little prior research has focused on simultaneously executing multiple services to improve reliability, as web services are already quire reliable and the additional costs of simultaneous executions cannot be justified by the expected reliability improvements [57].

Unlike web-based microservices, the ones executed in IoT and edge environments often suffer from partial failures and performance bottlenecks, as is expected for distributed execution environments with naturally dynamic and volatile resources. This work adapts the microservice architecture for such unreliable execution environments by systemically supporting the execution equivalence in microservice-based distributed applications.

The support for equivalence in existing microservice-based programming models [68, 77, 124] is limited: they either cannot explicitly express equivalent microservices or cannot efficiently execute them (i.e., minimize the resources consumed by executing workflows containing equivalent microservices). Without intuitive programming support for equivalence, a non-trivial development effort is required to cost-efficiently increase the reliability of a microservice-based application.

In this section, we describe a dataflow-based programming model that adds support for equivalence in orchestrations of microservice-based applications. Our programming model extends the dataflow programming pattern in [68]: programmers declaratively specify microservices and their dataflow relationships; the compiler automatically generates a workflow that schedules the execution plan for these microservices, with different execution strategies expressed as workflow constructs; and the runtime steers the execution of microservices based on the workflow and their execution results. In particular, we extend the dataflow specifications and workflow constructs with support for equivalent microservices, and provide rules to generate and execute such workflows. Our evaluation demonstrates that our solution simplifies the expression of equivalent functionalities and suites particularly well for adapting distributed executions to dynamic contexts.

As a summary, the contribution of this work is three-fold:

- We introduce a dataflow-based microservice orchestration language that explicitly supports execution equivalence.

- We introduce workflow constructs for executing equivalent microservices that provide a fine-grained control over the life cycle of microservice execution.

- Through case studies, we demonstrate how our solution can be applied to develop real applications, and how the resulting workflows can increase their reliability cost-efficiently. We also show that our solution outperforms prior approaches in striking the right trade-offs between reliability improvements and resource consumption.

5.1.1 Background and Related Work

This work focuses on providing programming support for orchestrating services containing equivalent microservices. We first discuss how microservice-based applications have taken advantage of equivalence, and then summarize major programming models for engineering such applications.

Equivalent Microservices Hosted at different cloud servers with dissimilar QoS characteristics, various microservices can provide the same functionality. In the research domain of cloud-based microservice composition, such equivalent microservices are referred to as competing microservices [158]. This domain focuses on how to choose a set of services that maximize the overall QoS while satisfying the QoS requirements of each service [3]. Hiratsuka et al. [57] further explore the combined use of functional-equivalent microservices to enhance the QoS. They leverage two general orchestration patterns for equivalent microservices: fail-over for reliability enhancement and speculative parallel for efficiency enhancement. In the edge and IoT domains, Osmotic computing [171] switches between cloud/edge-based microservice deployments to optimize the overall QoS.

In edge/IoT environments, equivalent microservices can also deliver the same functionality. However, these microservices can differ not only in their respective QoS characteristics, but also in the way they are implemented, including the hardware/software resource utilization, algorithms, and compositions. For example, [84] demonstrates that the environmental temperature can be captured by a temperature sensor, or be inferred from the CPU temperature; both wireless methods [145] and optical methods [114] have been used to obtain the indoor location of individuals. As an edge application is expected to run in dissimilar edge environments that feature different available sensor and computational resources and runtime contexts, it is hard to guarantee the overall reliability given the low reliability of individual microservice executions [148]. The combined use of equivalent microservices can improve reliability while striking a good balance between the response time and costs. Therefore, when extending the microservice architecture to the domains of IoT and edge computing, microservice equivalence can increase the power and expressiveness of existing programming models.

Programming Models for Orchestrating Microservices Workflow languages (e.g., WS-BPEL [124]) are widely used in engineering service-oriented systems and applications, due to their ease of use and ability to manage complexity [37]. A service/application workflow can be represented as a set of microservices and assist workflow control nodes, together with their order of invocation and data passing relationships [1]. In general, the assist workflow nodes consist of a start node, an end node, and any number of repeatable predefined workflow constructs. Such constructs represent different workflow control patterns (some researchers use different terms to represent the same concept, e.g., structured activities [124] or operational semantics [24]). At runtime, an execution engine runs workflows by following the operational semantics of the contained workflow constructs.

BPEL is a block-structured workflow description language that helps developers to express and execute workflows. The workflow patterns that can be expressed by BPEL are: sequential processing, conditional behavior (if), repetitive execution (while), selective event processing (pick), parallel processing and processing multiple branches (foreach) [124]. Accordingly, the supported workflow constructs are: sequence, parallel (AND-fork/join), exclusive choice (XOR-fork/join), loop, and multi-choice [137]. Although over 40 workflow patterns (including structured discriminator, which supports speculative parallel execution) have been developed, most of them are not explicitly supported in general workflow orchestration languages, such as BPEL [137].

The sheer number of BPEL features complicates the language's functional semantics. Besides, designed for machine processing, BPEL requires its developers to master graphical composition tools. Several novel domain-specific languages improve programmability. Orc [77] is a structured language, in which service orchestrations are specified by means of functional programming idioms. It also provides semantic support for handling concurrency, time-outs, exceptions, and priority. The workflow patterns in Orc (specified as combinators) support fail-over (`otherwise`) and speculative parallel (`pruning`) execution strategies. Although these strategies can be used to orchestrate the execution of equivalent operations, the language provides no facilities to control the fine-grained lifecycles of these operations.

Both BPEL and Orc require the programmers to specify the control logic of concurrency and failure handling. To further shift the burden of workflow orchestration from the programmer to the compiler, dataflow-based domain-specific languages are introduced for orchestrating workflows automatically [68]. Such DSLs enable the programmers to specify the data dependencies between microservices, and generate the workflow accordingly, for "data dependencies equivalent to scheduling" [71]. Dataflow programming has been widely adopted by IoT/edge computing [29, 46, 47, 131, 138] and stream processing [58]. However, these dataflow languages have no support for equivalence.

5.1.2 Problem Analysis

We start analyzing the problem domain by giving two example use cases that demonstrate how equivalent microservices can be used to satisfy reliability requirements. These use cases are a fire detection system and an offline digital store.

Use Cases of Leveraging Equivalent Microservices (1) Use Case 1, `fireDetection` : One important security task of smart homes is detecting fire. Flame sensors have been commonly used in private homes and office buildings for a long time [43]. However, in those cases in which a flame sensor is temporally unavailable or absent altogether, sensor data fusion can also accurately detect fire. For example, one alternative method can combine a temperature sensor and a camera (two most widely deployed sensors in smart homes) to detect fires [30]. Specifically, if both the smoke density level extracted from captured environment images and the room temperature captured by the sensor exhibit unusually high levels, these two conditions happening simultaneously indicate the presence of fire. Hence, we have two equivalent strategies for detecting fires: (1) read a flame sensor, and (2) (a) read a temperature sensor; (b) capture and process image. As fire detection must be both time efficient and reliable, the strategies (1) and (2) can be executed speculatively parallel to fulfill these requirements.

The use case above can be expressed modularly as individual microservices. Microservice `thresholdCheck` takes as input `firePossibility`, and returns a `boolean` value `isFireDetected` . Microservice `readFlameSensor` takes no input, checks the states of flame sensors, and outputs `confidence`, which can be used as `firePossibility`. Microservice `sensorFusion` takes as input `smokeDensity` and `temperature`, and outputs `firePossibility`. Microservice `getTemperature` queries the temperature sensor and outputs `temperature`. Microservice `getImage` captures images by camera, and outputs `imageUrl`. Microservice `inferSmokeDensity` takes as input `imageUrl`, processes the image and outputs `smokeDensity`. Fig. 5.1 lists the input/output relationships of these microservices.

(2) Use Case 2, `purchaseItemDetection`: In offline digital stores, a customer purchases merchandise by picking up items from shelves; upon exiting the store, the customer's credit card is charged for the purchases. An enabling technology for such stores is *purchase detection*,

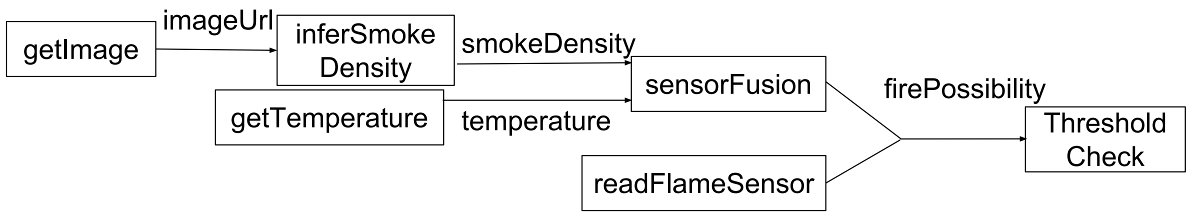


Figure 5.1: Data Dependencies of Use Case 1

using sensors to track purchases in real time.

Table 5.1: Microservices Used in `purchaseItemDetection`

Microservice	Input	Output
getBarcodeFromVideo	video	barcode
getItemIDFromBarcode	barcode	itemID
getShelfFromVideo	video	shelfID
estimateItemLocation	video, shelfID	row, line
getItemIDFromLocation	row, line, shelfID	itemID
getWeightChange	shelfID	weight
estimateItemIDbyWeight	shelfID, weight	itemID

The `purchaseItemDetection` application takes a video clip of a purchase as input and outputs the `itemID` of the purchased item. To recognize the purchased item, the application first analyzes the video clip for the item's barcode. If this method fails, it infers the `itemID` by processing sensor data in two equivalent ways: (1) use the shelf's id and the purchased item's location on the shelf to infer what the purchased item is; (2) obtain the delta of weight from the shelf's scale, and based on the delta infer what the purchased item is.

Deficiencies of Existing Programming Models Existing programming models lack explicit equivalence facilities and cannot support the above application scenarios.

(1) No semantic and syntactic facilities for equivalence in dataflow specification languages. An approach presented in [57] provides the pipeline, data distribution and data aggregation patterns. When translated to workflows, pipelines are converted to execute sequentially, and data distributions/aggregations are converted to execute in parallel (AND-fork/join). Hence, this dataflow specification has no semantic and syntactic support for the fail-over

and speculative parallel workflow patterns that leverage execution equivalence.

(2) Inability to handle exceptional execution conditions systematically: Consider removing the equivalent microservice `readFlameSensor` to generate use case 1' without any equivalent microservices. Fig. 5.2 shows how use case 1' can be expressed in a dataflow language. Notice that the dataflow contains no processing rules that specify how to handle execution failures. That is, when a microservice fails, no alternate microservice can be invoked to continue the execution, thus causing the overall execution to terminate.

```
//...binding microservices
getImage -> inferSmokeDensity
inferSmokeDensity -> smokeDensity
getTemperature -> temperature
(smokeDensity,temperature) -> sensorFusion
sensorFusion -> thresholdCheck
```

Figure 5.2: Dataflow-based Workflow Specification for Use Case 1'

However, with equivalence, the terminate-by-default failure handling rule no longer applies. For example, if the workflow of use case 1' is extended with an equivalent microservice `readFlameSensor`, the failure of `getTemperature` no longer terminates the overall execution.

(3) Poor cost efficiency: Although prior approaches, including Orc [77] and the“ 1 out-of-m join” [137] can express the execution strategies of equivalent microservices, these approaches were not designed to provide a fine-grained control over the life cycle of microservices. When executing workflows with equivalence, this lack of control may lead to using computational resource unproductively. Consider the following examples: 1) `readFlameSensor` finishes its execution at time t_1, while `getImage` and `getTemperature` remain in operation. To steer the execution cost efficiently would necessitate passing `firePossibility` returned by `readFlameSensor` to `thresholdCheck`, and immediately terminating both `getImage` and `getTemperature`. Although the “1-out-of-m join” pattern does execute `thresholdCheck`, it does nothing to stop the now unnecessary execution of `getImage` and `getTemperature`; 2)

`getTemperature` times-out at time t_2, while `getImage` remains in operation. As `sensorFusion` requires both `temperature` and `smokeDensity`, missing either one of these required inputs would make it impossible to execute `sensorFusion`. Hence, efficiency would necessitate terminating `getImage` and waiting for `readFlameSensor` to complete. However, Orc would wait for `getImage` to finish its execution and then proceed to executing `inferSmokeDensity`.

5.1.3 Workflow and DSL for Equivalence

We first introduce special equivalence-supporting workflow constructs and discuss how they are supported in the runtime. As a specific example of supporting equivalence programmatically, we discuss the design and implementation of our dataflow DSL.

Workflow Overview The main distinction of workflows with equivalence is that the successful/failed execution of microservices may affect the execution status of other microservices currently in operation or to be invoked. In traditional workflow graphs, nodes denote microservices, and a directed edge between them denotes their execution order. Although handling equivalence requires storing and executing additional control flows for each microservice, we maintain the basic structure of traditional workflows, adding to it new workflow constructs and notification rules. Unlike the operational semantics of general workflows, our new workflow control constructs collect the necessary execution states of the microservices within their scope and react accordingly.

Workflow Constructs A workflow graph, representing a service, contains a set of nodes as a set of directed edges, $G =< N, E >$.

Edges: A directed edge $e(n, m, d)$ connects node n to node m, $\forall n, m \in N$, and specifies the data d passed from n to m. We call n predecessor and m successor.

Nodes: $N = N_{ms} \cup N_{control}$, where N_{ms} denotes a set of microservice nodes and $N_{control}$ denotes a set of control nodes. $N_{control}$ contains one service start node *start*, one service end node *end*, and any number of pairs of workflow control nodes. For any microservice node in N_{ms}, $n(i, o, timeout)$ maintains its required input i, generated output o, and allowed execution time *timeout*.

Workflow Patterns: $C(k)$ denotes a pair of control nodes, with C_{start}, C_{end}, and k representing the start node, the end node and the start node's out-degree, respectively. A pair of control nodes can be one of the following three types:

- parallel, where k represents the number of concurrent branches. A parallel pair starts all k branches simultaneously, until all executions complete. If one branch fails, the pair fails, terminating the executions of the remaining branches.

- fail-over, where k represents the number of equivalent branches. It starts one branch at a time and outputs the results of the first successful execution.

- speculative parallel, where k represents the number of equivalent branches. It starts all the k branches simultaneously, and outputs the first obtained result. If one branch succeeds, the pair terminates the executions of the remaining branches.

Multi-threading and Execution State The workflow's execution can be described as a finite state automata of states and transitions [178]. We introduce the operational semantics from the perspective of states and transitions. In particular, we introduce how the states and transitions are combined with concurrency (thereafter, we use threads to demonstrate all general concepts).

A service's execution starts from the root thread. Multiple child threads can be spawned by and joined to one parent thread. Each spawned child thread maintains a parent handle, synchronized across all its siblings, used to notify the parent of whether the child's execution succeeded or failed. Each thread maintains a `counter` and a `currentNode`. The `currentNode` indicates the current node being executed by this thread. An executing thread can be terminated by its `currentNode` or interrupted by its parent thread, with the `currentNode` handling the interruption. The `counter` indicates the number of successfully executed branches if the thread is executing a parallel control pair, the current executed branch for a fail-over pair, and the number of failed branches for a speculative parallel pair.

Our workflow design possesses the following features:

- Each microservice node has only one direct predecessor and one direct successor. In other words, the control node pairs control all the spawning and joining concurrency actions.

- For a pair of control nodes, the start node's out-degree equals the end node's in-degree. Although the control nodes can be nested (e.g., a pair of control node is contained in another pair of control nodes), the number of spawned threads at the start node of the pair equals to the number of joined threads at the end node.

A node's execution status can be in one of the three states: `running`, `succeeded`, and `failed`. The *start* and *end* nodes can be only in the `running` state. The start nodes of control pairs can be in the `running` and `succeeded` states, while other nodes can be in any of the three states. A `running` state can transition to either `succeeded` or `failed`. In the `succeeded` or `failed` state, the `currentNode` runs as dictated by the operational semantics of its type, which comprises terminating the current thread, sending notifications to the parent thread, interrupting child threads, and setting the `currentNode` to another node.

Workflow Operational Semantics The semantics is implemented by following these state transition rules. If the current state is:

running: A microservice node starts the microservice's execution. The node transitions to the `failed` state if the execution fails or times out, and transitions to the `succeeded` state if the execution succeeds. The *start* node sets the `currentNode` to its successor, and *end* outputs the results.

The start node of a parallel pair and a speculative parallel pair set the `counter` to k (the number of branches), spawn k child threads, set the `currentNode` of the child threads to the first nodes of these threads, and transition to the `succeeded` state (see Fig. 5.3.a and Fig. 5.5.a). The start of a fail-over pair sets `counter` to k if it equals 0, executes the *counter*th branch, and transitions to `succeeded` (see Fig. 5.4).

The end node of a control pair waits to be signaled by its child threads. Upon receiving a `succeeded` signal, 1) the end of a parallel pair decrements the `counter`, and transitions to the `succeeded` state if `counter` $==0$, or maintains the `running` state if `counter`>0 (see Fig. 5.3.b); 2) the end of a fail-over pair or a speculative parallel pair transitions to the `succeeded` state; Upon receiving a `failed` signal, 1) the end of a parallel pair transitions to the `failed` state; 2) the end of a fail-over or speculative parallel pair decrements the `counter`, and transitions to the `failed` state if `counter`$==0$ (see Fig. 5.5.b). If otherwise `counter`>0, the end of a fail-over pair sets the `currentNode` to its pair start (see Fig. 5.4), and the end of a speculative parallel pair maintains the `running` state. If the thread is interrupted, the end node of its control pair interrupts all child threads in turn.

succeeded: 1) A microservice node sets the `currentNode` to its direct successor. If the direct successor is the end node of a control pair, the thread sends a `succeeded` signal to its parent and terminates itself. Otherwise, it continues to execute the new `currentNode`. 2) the start node of a control pair sets the `currentNode` to its pair end node. 3) a pair end node sets `counter` to 0, interrupts all child threads if the pair is speculative parallel, and follows the microservice node's processing rules.

failed: 1) A microservice node checks if it is executed by the root thread: if so, it sets the `currentNode` to the end node; otherwise, the thread sends a `failed` signal to its parent and terminates itself. 2) an end node of a control pair sets the `counter` to 0, interrupts all child threads if the pair is parallel, and follows the microservice node's processing rules.

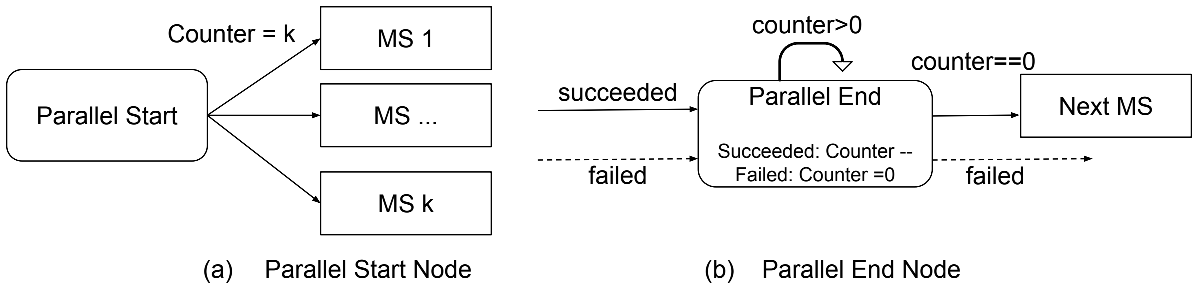


Figure 5.3: Parallel Pair State Transition

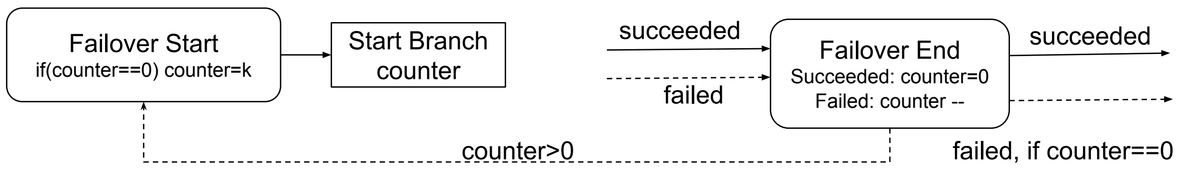


Figure 5.4: fail-over Pair State Transition

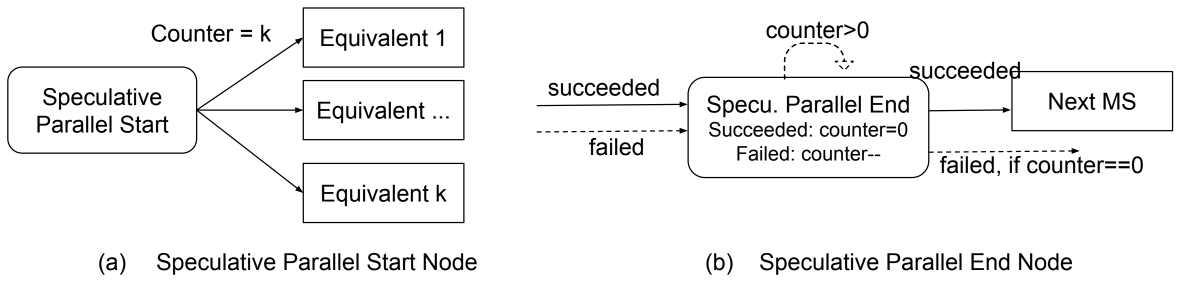


Figure 5.5: Speculative Parallel Pair State Transition

DSL and Workflow Generation To demonstrate how dataflow specifications can support equivalence, we create a domain specific language, MDLE (**M**icroservice **D**ataflow **L**anguage with **E**quivalence). A MDLE script declaratively specifies a collection of microservices. Aliases implicitly define the data flow between microservices. Multiple microservices providing the same input are considered equivalent. **MDLE EBNF:**

```
<Service> ::= <ID> <Description>
<ID> ::= "Service "String
<Description> ::= "{"[<Params>] <MSs>"}"
<Params> ::= "input:"|"output:" [<Variable> ","]
<MSs> ::= [Microservice]+

<Microservice>::="MS:"<MSID>"{" [<MSDetail>]+ "}"
<MSID>::=String

<MSDetail>::=<Timeout>|<Input>|<Output>|<Prior>
<Timeout>::="timeout:" [<Select_Rule> "."]+
<Input>::="input": [<MS_input> ","]+
<Output>::="output": [<MS_output> ","]+
<MS_output>::=<output_Variable> ["as" <Alias>]
<MS_input>::=[<Alias> "as"] <input_Variable>
<Alias>::=String
<Prior>:="priority:" "high"|"low"|"medium"
```

Figure 5.6: DSL EBNF Definition.

Fig. 5.6 defines the syntax of MDLE in EBNF. Some of the key features are as follows:

- Each service is identified by a unique id, `ID`. `Params` can either be `input`, which must be passed when the service is invoked, or `output`, which is the returned execution result.

- A service comprises `Microservices`, identified by unique `MSIDs`, and containing additional attributes.

- A microservice invocation comprises the following attributes: 1) the `Input` parameters that specify the microservice's invocation parameters; 2) the `Output` that specify what results should be returned, which can be renamed to `Alias`; 3) the `Priority` of a microservice, which can be `high`, `medium`, or `low`. Programmers can use the priority parameter to indicate which equivalent microservice should be preferred to provide the required input; 4) the optional `timeout` rules that specify the timeout values for each microservice. If the value is not specified, a default timeout value is used.

Fig.5.7 shows an example MDLE script. The original outputs of microservices A, B, C are a, b, c, and are aliased to x. Microservices A, B, and C are equivalent, and unless

```
1  Service example {
2     output: y
3     MS: A { output: a as x
4        //priority: medium}
5     MS: B { output: b as x
6        //priority: medium}
7     MS: C { output: c as x
8        //priority: high}
9     MS: D {
10       input: x; output: y}}
```

Figure 5.7: Example Service Suite

explicitly specified, their priorities are medium by default. Hence, they should be executed in a speculative parallel way. If line 4, 7 and 10 are uncommented, microservice C and the speculative parallel of A and B should be orchestrated as fail-over.

Aliasing Output and Input: A microservice may require multiple inputs, and can generate multiple outputs. To differentiate these inputs and outputs, a microservice developer assigns different names to these inputs and outputs. In a service script, these names are identified as `input_Variable` and `output_Variable`. MDLE uses `alias` to implicitly specify equivalent microservices. Two or more microservices are considered equivalent if their output is set to the same `alias`, which is a required input for another microservice.

Compiling a MDLE Script to a Workflow Graph: The MDLE compiler converts dataflows into a workflow graph, via a bottom-up procedure. The compiler maintains a dynamic set of nodes to process. The end node is inserted into the set first. Each node in the set is processed in turn, with the newly added nodes replacing the processed ones: 1) if the processed node requires more than one input, the parallel start and end nodes are added to the graph, with each required input becoming a special single-input branch node; 2) if multiple microservices can provide the input of the current node, a pair of corresponding execution control nodes are added, so the data providing microservices become the branch nodes of these control nodes; 3) if only one microservice provides the required input of the current node, it is added to the graph directly. While adding these nodes to the graph, if the current node already has an incoming edge, the new nodes are added between the edge's source node and the current node. The graph generation algorithm terminates once the dynamic set is empty. If the same microservice is invoked in all branches of a control pair, while being directly connected to the pair's start node, the microservice is removed from all branches and added before the pair's start node.

5.1.4 Evaluation

We start with a case study of generating and executing a workflow. Then, we assess how our solution improves reliability and efficiency, as compared with workflows without equivalence and those without fine-grained lifecycle control.

Case Study Continuing with the aforementioned use case, Fig. 5.8 shows the MDLE source code of the `fireDetection` service. Our workflow compiler and runtime are implemented in Java. Figs. 5.9 and 5.10 show the generated workflows of `fireDetection` and `purchaseItemDetection`, respectively.

Our execution parameters are 80% for the microservice execution success rate and a random number from 1-500ms for the microservice execution time. To demonstrate how the runtime works, we analyze the trace of one execution.

For use case 1, the 'Speculative Parallel Start' node starts executing at 1ms by forking two threads to execute the 'Parallel Start' node and `readFlameSensor`. At 4ms, the 'Parallel Start' node forks two threads to execute `getTemperature` and `getImage`. At 318ms, `getImage` finishes its execution, with `inferSmokeDensity` continuing on the same thread. At 471ms, `readFlameSensor` finishes its execution, passing a `succeeded` signal to the 'Speculative Parallel

```
Service fireDetection {
   output:isFireDetected
   MS: readFlameSensor {
      output: confidence as firePossibility
      priority: medium
   }
   MS: SensorFusion {
      input: smokeDensity, temperature
      output: firePossibility
      priority: medium
   }
   MS: getTemperature {output: temperature}
   MS: getImage {output: imageUrl}
   MS: inferSmokeDensity {
      input: imageUrl
      output: smokeDensity
   }
   MS: thresholdCheck {
      input: firePossibility
      output: isFireDetected
   }}
```

Figure 5.8: Source File of `fireDetection` Service Suite

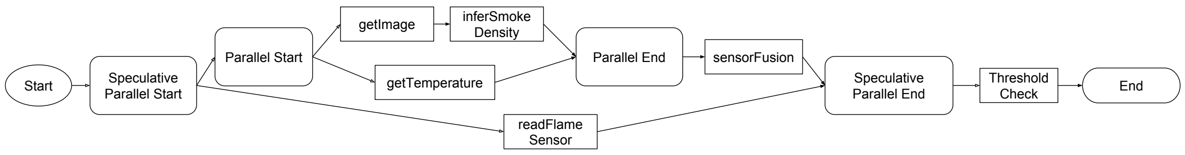


Figure 5.9: Generated Workflow for Use Case 1

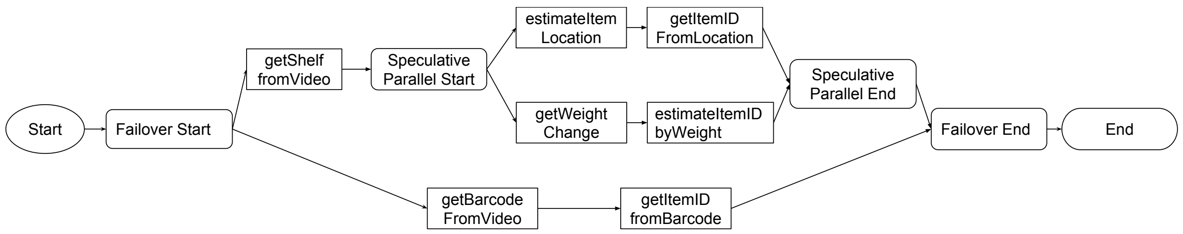


Figure 5.10: Generated Workflow for Use Case 2

End' node, waiting on the main thread. Upon receiving the signal, the 'Speculative Parallel End' interrupts its child threads, and then executes `thresholdCheck`. Upon receiving the interrupt, the 'Parallel End' node further interrupts its child threads, thus terminating the execution of `inferSmokeDensity`. At 722ms, `thresholdCheck` finishes its execution, and the

Table 5.2: Average Results of 1000 Runs with Varying Execution Time

avg execution time	100	200	300	400	500
successful rate	0.714	0.706	0.695	0.721	0.687
ms execution	380	719	1072	1444	1845
finish time	226	433	652	844	1139

’End’ node outputs the result of `thresholdCheck`.

For use case 2, the ’Start’ node transitions to the ’fail-over Start’ node, which sets the `counter` to 2 and the `currentNode` to ’fail-over End’, spawning a new thread to execute the second branch, `getBarcodeFromVideo`. The microservice fails at 203ms, terminating its thread and sending a `failed` signal to ’fail-over End’, which transitions to the ‘fail-over Start’ node to execute the first branch. At 327ms, `getShelfFromVideo` finishes its execution, and the connected ‘Speculative Parallel Start’ node spawns two threads. At 790ms, the weight change based approach finishes its execution and sends a `succeeded` signal to ‘Speculative Parallel End’, which passes the output to the end node and terminates its child threads, still executing `estimateItemLocation`.

Reliability and Cost efficiency To evaluate the reliability and cost efficiency of our workflow framework, we first set the microservice success rate to 0.8, while varying the average execution time between 100, 200, 300, 400, and 500 (ms). Table 5.2 shows the results of 1000 runs for each parameter combination. We observe that the overall successful rate is almost stable. The overall execution (i.e., cost) and completion times are proportional to the average microservice execution time.

Then, we set the average microservice execution latency to 200ms, varying the microservice success rate between 0.2, 0.4, 0.6, 0.7, 0.8 and 0.9. Table 5.3 shows the results of 1000 runs. With the increase of the microservice success rate, the overall reliability and the overall execution time increase. The successful execution of the first microservice causes additional

Table 5.3: Average Results of 1000 Runs with Varying Reliability

reliability	0.2	0.4	0.6	0.7	0.8	0.9
ms execution	542	600	680	710	750	753
finish time	379	398	441	431	443	424
successful rate	0.034	0.193	0.363	0.543	0.718	0.872

microservice invocations.

We further compare our use case 1 solution with two alternatives: 1) `without equivalence`—randomly execute one of the two equivalent methods; 2) `without terminating`—execute the generated workflow graph without terminating any microservices in operation. We set the microservice reliability to 0.8 and the average latency to 100ms, repeating the simulation 1000 times. Table 5.4 shows that in comparison to `without equivalence`, our solution improves the reliability by 45.9%, reduces the completion time by 24.7%, with the cost of the overall microservice execution time (which can be taken as the resource cost) increasing by 37.7%. Compared to `without terminating`, our solution reduces execution time (i.e., cost) by 27.3%.

We compare use case 2's execution results with those of the aforementioned two alternatives, parameterized identically. Table 5.5 shows that compared with `without equivalence`, our solution improves the reliability by 60%, while the overall cost and completion time increase by 11.4% and 3.8%, respectively. Compared to `without terminating`, our solution reduces the execution time (i.e., cost) by 3.9%.

The results of both use cases show that our solution enhances the reliability of microservice-based applications. Due to its fine-grained lifecycle execution control, our solution eliminates the costs of executing microservices that have become unnecessary, a particularly effective optimization for the parallel and speculative parallel execution patterns.

Table 5.4: Comparison among Three Solutions for Use Case 1

	successful rate	execution time	finish time
our method	0.706	380	226
without equivalence	0.484	223	300
without terminating	0.706	523	226

Table 5.5: Comparison among Three Solutions for Use Case 2

	successful rate	execution time	finish time
our method	0.888	308	259
without equivalence	0.555	276	250
without terminating	0.888	320	259

5.1.5 Discussion

Based on the evaluation results, we revisit some of the design decisions behind our workflow and MDLE.

Supported Workflow Constructs Although our workflow lacks the “if-else” switch and the “while” loop control patterns, in line with other dataflow-based DSLs [68], we discuss how they can be added to the workflow and MDLE. Adding the “while” loop to the workflow can be treated as a special variant of sequential execution, without spawning any threads. The the switch branches of “if-else” can join in one microservice. For example, to identify a person in a video: if a face image is detected, invoke a face recognition microservice; otherwise, invoke a gait recognition microservice. With the two branches in a “if-else” switch generating the same output, the runtime can execute these switches sequentially. A dataflow-based DSL can encapsulate the “while” conditions within microservices and express the “if-else” switch by conditionally aliasing microservice outputs.

Syntactic Support for Equivalence To support equivalence, programmers must specify: 1) which microservices are equivalent; and 2) how to orchestrate their execution. A dataflow-based DSL can use other alternatives as well. For example, $a * b - c$ can denote that a, b, c

are equivalent and orchestrated to execute a, b first speculatively parallel and then execute c if both a, b fail.

However, we choose aliasing and priority to implicitly denote equivalence and orchestrations as: 1) an alias has a unique meaning throughout an application. Aliasing the output in a microservice clearly expresses that the output has its correct physical meaning, thus avoiding programming errors; 2) in the presence of multiple equivalent microservices, programmers only have to decide which microservice's QoS features express the requirements, without having to explicitly orchestrate microservice execution. Our design shifts the burden of orchestrating equivalent microservices from the programmer to the compiler.

5.1.6 Conclusion

We add programming support for equivalence in microservice-based applications by introducing a dataflow-based DSL that extends the notion of dataflow with declarations of equivalent microservices and their execution patterns. Our new equivalence workflow constructs enable the automatic generation of reliable and efficient microservice execution workflows. Supporting equivalence enhances the reliability of microservice-based applications, while our workflow design enhances their cost efficiency.

5.2 Workflow Meta-Pattern for Equivalent Microservices

One of the major challenges in provisioning mobile services is achieving *QoS-optimality* [41, 168]. Because edge and IoT environments are less dependable than traditional cloud environments, mobile services are often unreliable and untrustworthy. When it is mobile and energy-harvesting devices that provide edge resources, the resulting services become vulnerable to partial failure and low reliability [41, 168]. Besides, operated in physically unprotected environments, devices can be compromised to report false information, so the services they provide become untrustworthy [31]. Moreover, the resource constraints of these devices render their services more sensitive to execution cost and latency.

The microservice architecture isolates business functionalities into fine-grained building blocks [55], and applies workflow patterns [165] to assemble the resulting microservices into services. Microservices are considered *equivalent* if they satisfy the same requirements by different means (e.g., authenticating a user via a password, biometrics, SMS, or touchscreen patterns). The reliability and trustworthiness of mobile services can be improved by exploiting *the combined execution of equivalent microservices* [15, 153].

Workflow patterns describe common execution strategies that solve recurrent problems in process-oriented applications. Workflow patterns that describe combined executions of equivalent microservices include failover, speculative parallel, and majority voting. These workflow patterns provide the same functionality, while enhancing certain QoS characteristics (e.g., speculative parallelism for performance, failover for reliability, and majority voting for trustworthiness). However, intended for a small number of equivalent microservices, these workflow patterns' crude-grained execution strategies cannot achieve optimal QoS for larger microservice numbers. Consider improving accuracy: with up to several equivalent microservices, majority voting improves accuracy without incurring unreasonable microservice usage fees. However, with a larger number of equivalent microservices, their usage fees to execute this pattern can become prohibitive.

As compared to coarse-grained patterns, fine-grained patterns can balance QoS characteristics better. For example, to better balance cost and accuracy as compared to majority voting, some equivalent microservices can be executed first, with their results' coherence examined to determine whether to execute the remaining ones [15]; to improve reliability while controlling for costs and latency, approximation algorithms are applied to discover an optimal execution strategy [20, 57].

However, fine-grained workflow patterns are hard and error-prone to express, implement, and maintain. Customizing workflow patterns in general-purpose programming languages is tedious, as it requires synchronizing multi-threaded execution and data exchanges. A fine-grained pattern can also be formed by nesting crude-grained patterns, but this approach suffers from several drawbacks: 1) some fine-grained patterns need to access the execution results of all constituent microservices, while some crude-grained patterns may not output their intermediate results; 2) nesting workflow patterns is hard to express and understand (i.e., to elucidate a nested workflow pattern, workflow expressions are accompanied by flow charts [20]).

This work introduces a workflow meta-pattern that declaratively specifies fine-grained workflow patterns for the combined execution of equivalent microservices. In particular, our meta-pattern describes a fine-grained workflow pattern as 1) an algebraic expression that denotes the invocation sequences of equivalent microservices and 2) a Boolean function that determines whether to terminate the execution. To demonstrate how our meta-pattern can concisely and flexibly express the combined execution of equivalent microservices on different programming platforms, we implemented it as a Scala library and a YAML-based DSL. We further integrated the resulting workflow patterns into realistic mobile services. We evaluated these integrations to determine how effectively our approach optimizes the QoS of mobile services and how much programmer effort it requires as compared to the state of the art.

The contribution of this work is three-fold:

(1) We introduce a meta-pattern for declaratively specifying fine-grained patterns that describe the combined execution of equivalent microservices to improve QoS.

(2) We concretely implement our meta-pattern in Scala and YAML to provide programming support for composing QoS-optimal mobile services.

(3) We apply our reference implementation to compose several practical mobile services and empirically evaluate their QoS characteristics in dissimilar execution environments.

5.2.1 Background

We introduce workflow patterns as well as mobile and IoT services, background required to understand our contribution.

Workflow Patterns In SOA, workflow patterns serve as basic building blocks. Based on their application targets, workflow patterns divide into multiple categories: control flow, resource, data, and error handling patterns. We use the following control flow [179] to introduce a meta-pattern for generating fine-grained workflows for the combined execution of equivalent microservices:

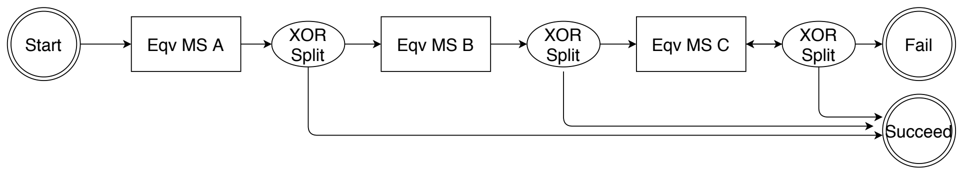
- XOR Split: connects to multiple microservices that can be invoked. Only one branch executes given a condition.
- AND Split: connects to multiple microservices that can be invoked, with all branches executing in parallel.
- AND Join: connects from multiple microservices; the following process continues only upon receiving all results.

- Cancelling Discriminator: connects from multiple microservices; the following process continues upon receiving any result, with the remaining branches terminated.
- Cancelling Partial Join (a.k.a, M-out-of-N join): connects from N microservices; the following process continues upon receiving M results, with the remaining branches terminated.

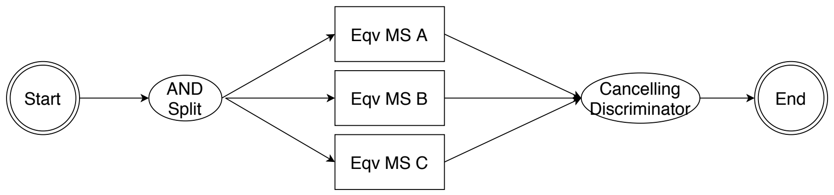
Existing Patterns for Equivalent Microservices Several well-known workflow patterns describe the combined execution of equivalent microservices. As shown in Fig. 5.11, *failover* [2] improves reliability by switching to equivalent microservices upon failure; *speculative parallel execution* [82, 153] executes multiple equivalent microservices simultaneously and uses the first result to improve both reliability and latency; *majority voting* [185] compares the execution results of multiple equivalent microservices and outputs the mostly likely result to improve trustworthiness.

Mobile and IoT services As edge computing technologies evolve, mobile and IoT services become possible [29, 131]: mobile and IoT devices at the edge expose their sensing and computing capabilities as services, accessed by nearby client devices.

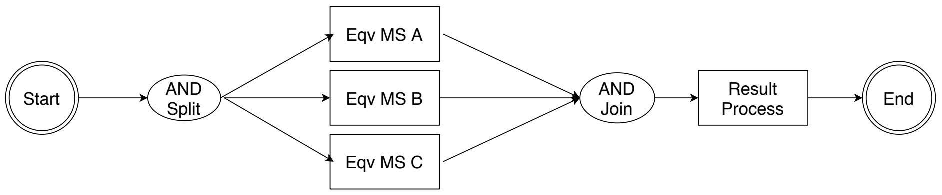
Equivalent microservices are common in mobile environments: 1) *recognizing facial images using services provided by different vendors* [15]; 2) *authenticating users* by means of a password, biometrics (fingerprint, iris or facial image), SMS, or touchscreen patterns [108, 135]; 3) *detecting atmospheric particulate matter value (PM2.5)* by reading from a portable PM2.5 sensor, estimating from images [100], or invoking the web service of the nearest environmental station; *detecting crowds* by reading a weight sensor, recognizing persons from the area's camera image[92], using an entrance-exit counting device, or counting WiFi beacons[143].



(a) the Fail-over Pattern



(b) the Speculative Parallel Pattern



(c) the Majority Voting Pattern

Figure 5.11: Combined Execution of Equivalent Microservices A, B, and C

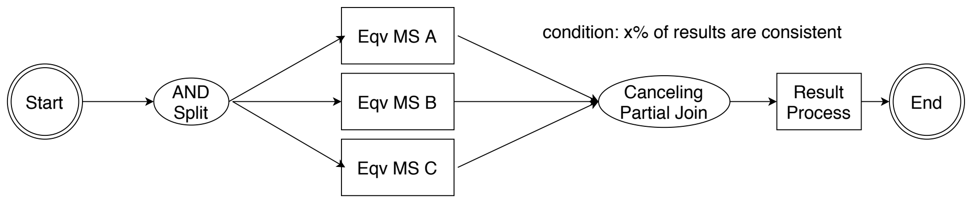
When provisioning mobile services, *QoS-optimality* is hard to achieve [41, 168]. The mobility and diversified ownership of these devices lead to low reliability and trustworthiness. The combined execution of equivalent microservices can improve these QoS characteristics. However, compared with web service composition, mobile services pose two unique challenges: 1) the variety and number of equivalent mobile services are significantly larger as compared with web services, as data-rich mobile environments feature multiple ways to satisfy the same requirement while microservices provided by different mobile devices are also considered equivalent [152]; 2) mobile services are provided in resource constrained environments, rendering them more in need of QoS optimality. Next, we demonstrate by example how a larger number of equivalent microservices requires a fine-grained workflow pattern to optimize service QoS.

5.2.2 Motivating Scenario

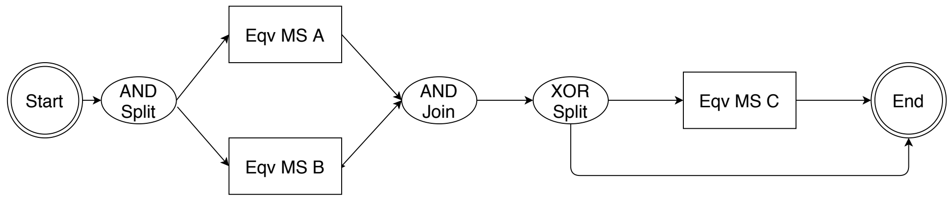
Authentication, expression analysis, and emotion recognition rely on detecting and locating faces in images and videos. Facial detection are provided as web services by different vendors and as deployable mobile services [15]. However, none of these equivalent microservice is 100% accurate, as the quality of input images and videos affect the accuracy of these functionalities [74]. To improve accuracy, the majority voting pattern has been applied [172], which executes all alternatives simultaneously, and waits till receiving all results to determine the final output. We use facial detection as an example to demonstrate how fine-grained patterns improve overall QoS as compared with majority voting and the problems in constructing such patterns.

Fine-Grained Patterns for Optimizing QoS Alas, majority voting improves accuracy at the expense of increasing execution latency and cost: the additional latency is incurred by the necessity to wait for those microservices that takes longer to execute, while the additional cost is incurred by the necessity to invoke all equivalent microservices. In the presence of equivalent microservices whose execution latency or cost is unusually high, a more fine-grained workflow pattern can better optimize the QoS of the combined execution of equivalent microservices.

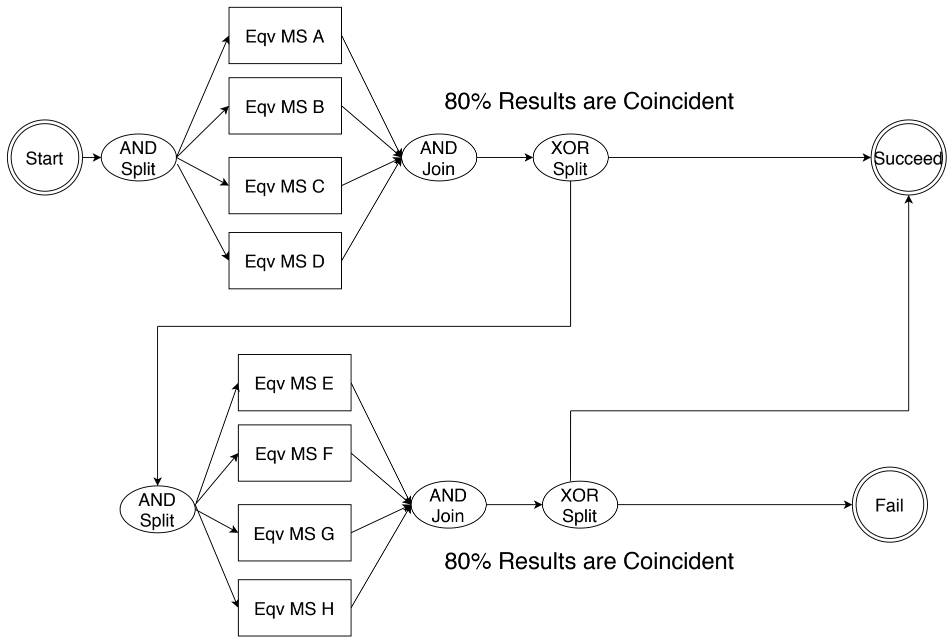
To demonstrate how fine-grained workflow patterns work, we denote three equivalent microservices as “A”, “B”, and “C”. To increase accuracy while reducing the overall execution latency, **example pattern 1** executes “A”, “B”, and “C” simultaneously, and terminates upon receiving two coincident results. If the first two results are the same, the execution can terminate without waiting for the third result, which could incur unusually high latency. To increase accuracy while reducing the overall cost, **example pattern 2** first executes “A” and “B” simultaneously, and waits for both of their results. If “A” and “B” return the same result, output it as final; otherwise, execute “C” and output the results agreed upon by any two microservices.



(a) Fine-grained Pattern 1: Optimized for Latency Efficiency



(b) Fine-grained Pattern 2: Optimized for Cost Efficiency



(c) Fine-grained Pattern 3: 8 Equivalent Microservices

Figure 5.12: Fine-grained Combined Execution of Microservices A, B, and C

Problems with Expressing Fine-Grained Patterns Fig. 5.12 demonstrates how workflow constructs can express the aforementioned fine-grained patterns. For the pattern in Fig.

5.12.a, we change the semantics of the standard construct “M-out-of-N join” from “terminating upon receiving M results from N branches” to “terminating upon a certain condition,”, i.e., “two received results coincide” in our case.

Although standard pattern constructs can fully support the pattern in Fig. 5.12.b, the required number and complexity of workflow constructs would be much higher than in the standard majority voting pattern. As the number of equivalent microservices grows, expressing such patterns would become unwieldy. For example, **example pattern 3** can be: for 8 equivalent microservices, execute every four in a row, and continue to execute the next row of four microservices only if less than “80%” of previous results coincide. It takes 6 workflow constructs to express this pattern, while the condition of “80% of all results coincide” needs to be repeated twice (Fig. 5.12.c). To make things worse, the “XOR split” needs the execution results of all equivalent microservices to determine the next step, while these results serve as intermediate information and are not exposed to these external “XOR split” constructs.

The necessity to change the semantics and data access of basic workflow constructs makes nesting workflow constructs tedious and error-prone, while unwieldy nested workflow patterns are hard to understand and maintain. These shortcomings motivate the need for dedicated programming support for the fine-grained combined execution of equivalent microservices.

5.2.3 Meta-pattern Design and Implementation

To express and manage the combined execution of equivalent microservices, we design a meta-pattern that generates workflow patterns with the following properties:

1. Applicable to microservices that are equivalent;

2. The generated workflow shares the same input and output with its constituent microservices, thus providing the same functionality;

3. Compared with its constituent microservices, the generated workflow improves at least one QoS characteristic.

Next, we introduce the syntax, semantics, design rationale, and visualization of our meta-pattern, as well as its applicability and runtime support.

Meta-Pattern Syntax and Semantics Formally, the meta-pattern expresses a pattern as a triple $m =< \theta, \zeta, \omega >$:

- θ: a set of equivalent microservices;
- ζ: an invocation sequence, an expression that denotes the complete execution order of the equivalent microservices;
- ω: a terminating condition, a Boolean function that takes as input the receive results of microservices' executions and outputs whether to terminate the workflow execution.

For example, the workflow pattern in Fig. 5.12.b can be expressed by Fig. 5.13. It describes a sequence of "executing A and B in parallel first (i.e., $A * B$), and then C (i.e., $-C$)", which can be short-circuited upon reaching the condition: the mostly agreed upon result should reach at least 60% of all received votes.

```
m_1=< θ = (A, B, C),
    ζ = A*B-C,
    ω = mostVotedResult.votes/totalVotes>=0.6 >
```

Figure 5.13: Meta-Pattern for Expressing Fine-Grained Pattern 2

An invocation sequence is expressed by a set of equivalent microservices and the operators connecting them into an expression. The binary operators $-$ and $*$ denote a sequential and a parallel execution, respectively. For example, given two equivalent microservices a and b, $a-b$ expresses that the microservices are to be executed in sequence from left to right, while $a*b$ expresses that the microservices are to be executed in parallel. Notice that because the $-$ and $*$ operators take equivalent microservices as their operands, the traditional built-in operator precedence is slightly altered. For example, for the invocation sequence $a-b*c$, a is executed first; then b and c are executed in parallel. The parentheses operators denote that the invocation sequence inside a pair of parentheses is considered as one equivalent microservice. For example, $a*b-c$ means to execute a and b in parallel first and then c, while $a*(b-c)$ means to treat $b-c$ as an equivalent microservice, and execute a and $b-c$ in parallel. Fig. 5.14 gives the EBNF grammar of an invocation sequence.

```
invokeSeq(ζ) ::= f|(f)|f-f|f*f, ∀f ∈ θ
```

Figure 5.14: EBNF Definition for Invocation Sequence Specification

An invocation sequence can terminate at different points between runs. In the example in Fig. 5.13, "C" would be executed only of "A" and "B" return different results. Terminating conditions control such variability across runs.

Design Considerations Our meta-pattern for the combined execution of equivalent microservices describes a workflow pattern as an invocation sequence of microservices that is short-circuited upon reaching a specified condition. Two observations inform our design:

1. For the combined execution of equivalent microservices, different terminating conditions determine which QoS characteristic to enhance, while different invocation sequences determine how to balance the remaining QoS characteristics. Common terminating conditions include: any results is received (to enhance reliability) and the received results coincide (to enhance trustworthiness). The equivalent microservices

are executed either in parallel or in sequence; the parallel execution incurs additional cost but shortens latency, and the sequential execution is vice versa. To optimize service QoS, one can vary the terminating conditions and the invocation sequences of equivalent microservices.

2. An invocation sequence cannot be altered, only continued or discontinued. The generated patterns only apply to equivalent microservices, so a microservice's result cannot serve as a control flow condition that determines which microservice to execute next. Hence, an invocation sequence is expressed with no control flow constructs, with "-" and "*" denoting sequential and parallel invocations, respectively.

Specifying and Visualizing Patterns We first reflect on three intrinsic properties of invocation sequences, and then describe how these properties inform the rules for expressing patterns. Finally, we show how patterns can be depicted as flowcharts.

Observation 1: The parallel operation is commutative, while the sequential one is not, e.g.: $a * b = b * a$, while $a - b \neq b - a$.

Whether two invocation sequences are equivalent depends on whether they express the same execution control logic. $a * b$ means to execute a and b in parallel, while $b * a$ also means to execute a and b in parallel. In contrast, $a - b$ means to execute a first and then b, while $b - a$ means to execute b first and then a. Hence, $a * b = b * a$, while $a - b \neq b - a$.

Observation 2: Both the parallel and sequential operators are associative, e.g.: $a - b - c = (a - b) - c = a - (b - c)$, and $a * b * c = (a * b) * c = a * (b * c)$.

$a - b - c$ means to execute a first, then b, and then c. $(a - b) - c$ and $a - (b - c)$ express exactly the same execution control logic. The same argument applies to $a * b * c$, $(a * b) * c$ and $a * (b * c)$.

Observation 3: Parenthesis are only required to disambiguate expressions that contain the "$-$" operator (not nested in other parenthesis), with the "$*$" operator appearing right before or after the expression's parenthesis. E.g.: $(a - b) * c \neq a - b * c$, $(a * b - c) * d \neq a * b - c * d$, while $a - (b * c) = a - b * c$.

There are three different possible operator combinations inside and outside parenthesized expressions: (1) no un-nested $-$ inside, e.g., $(a * b) - c$, $a - (b * c) - d$, $(a * b) * c$, $a * (b * c) * d$, $a - (b * c) * d$, or $\big((a - b) * c\big) - d$ (the outside parenthesis); (2) an un-nested $-$ inside, with no direct connections to $*$ right outside, e.g., $(a * b - c) - d$, or $a - (b - c) - d$; (3) an un-nested $-$ inside, with at least one connected $*$ right outside, e.g., $(a - b) * c$, $a * (b - c) - d$, or $a - (b - c) * d$. The parentheses in (1) and (2) can be removed based on observations 1 and 2 above. However, the parentheses in (3) cannot be removed, as $(a - b) * c$ and $a - b * c$ expresses different execution logic: $(a - b) * c$ executes a and c in parallel, and will not execute b unless a returns a failure before c successfully returns; $a - b * c$ executes a first and then b and c in parallel.

Based on these observations, the following conventions should be followed to ease readability and maintainability:

1. If equivalent microservices can be switched without altering the semantics, an invocation sequence should list the microservices in the alphabetical order of their names.
2. An invocation sequence should contain only irremovable parentheses (see Observation 2 and 3).

Visual programming interfaces commonly integrate workflow patterns. To ease the integration of the generated patterns, we introduce the following visual abstractions to express an generated pattern: 1) an "Eqv Start" Node that connects to all equivalent branches, specifying the invocation sequence (i.e., $M.\zeta$); 2) a set of equivalent branches ($M.\theta$); 3) an "Eqv

End" Node that connects from all equivalent branches, specifying the terminating condition ($M.\omega$). Fig. 5.15 demonstrates how these abstractions visualize the aforementioned example pattern 3. To express this fine-grained pattern, our visual abstractions are more flexible and concise than those offered by the nested workflow patterns shown in Fig. 5.12.c.

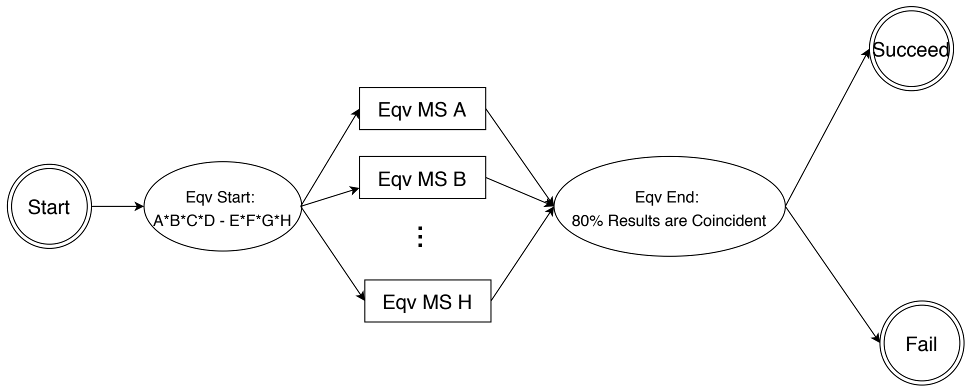


Figure 5.15: Expressing Fine-grained Pattern 3 With Meta-Pattern

Runtime Support for Executing Patterns Executing the workflow patterns specified by our meta-pattern requires dedicated runtime support. The flowchart in Fig.5.16 explains the execution logic for this runtime support:

- S_0: to start executing, a workflow pattern receives input parameters and then transitions to state S_1, "execute next equivalent microservices".

- S_1: based on ζ, determine which microservices to execute next and initialize them, transitioning to state S_2, "waiting for any results to be returned".

- S_2: wait to receive any microservice execution result or for the overall execution to timeout. Upon timeout, transition to the"failure" state. Upon receiving a result, persist it with the other microservice execution results, transition to state S_3, "applying the terminating condition."

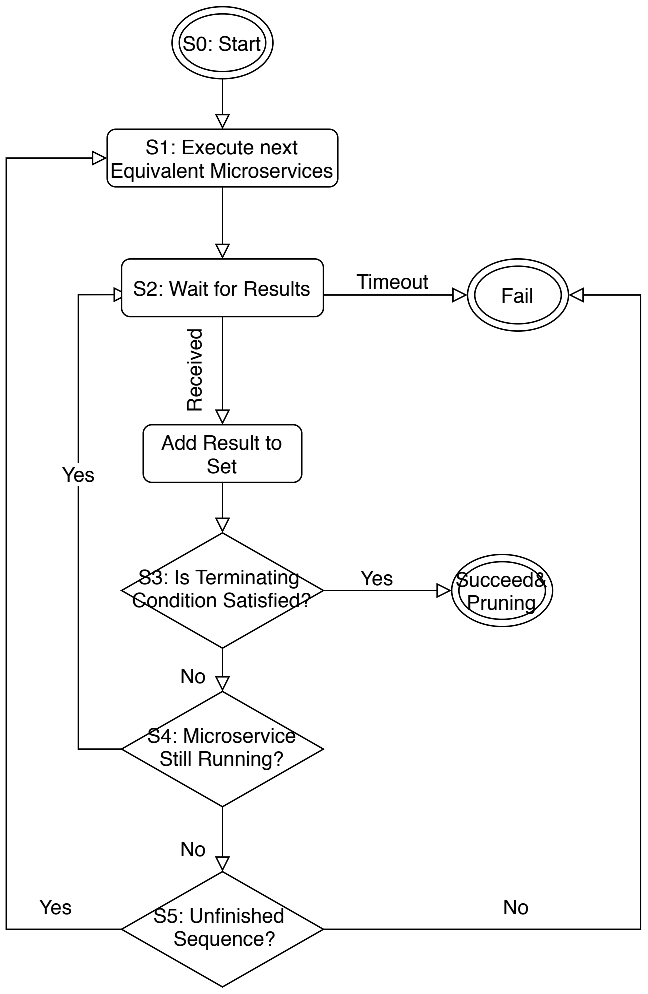


Figure 5.16: Runtime Support for Executing Generated Patterns

- S_3: apply the terminating condition ω, output a Boolean value indicating whether to terminate the execution. If true, transition to the final state,"success and pruning", which terminates all unfinished microservices and outputs the final result; Otherwise, transition to state S_4, "checking whether there is any microservice running."

- S_4: If no microservices are still running, transition to state S_5, "checking if the invocation sequence has reached the end;" otherwise, transition to state S_1.

- S_5: check if there are still microservices in ζ waiting to be executed. If true, transition to state S_0; otherwise, transition to the "failure" state.

5.2.4 Reference Implementation and Evaluation

We implement our meta-pattern design as a Scala library for functional programming and a YAML DSL for edge services, respectively. We demonstrate that the meta-pattern is expressive enough to generate fine-grained workflows that enhance the performance of mobile/IoT services. Our evaluation shows that compared with crude-grained patterns, the generated fine-grained patterns improve the overall QoS.

Reference Implementations **Scala:** The Scala-based reference implementation (Scala SDK 2.12.8) comprises approximately 870 lines of code (ULOC). To allow any set of functions with the same signature to represent equivalent microservices, the library features a generic function container and an invocation sequence class. Each constituent function is wrapped into a container object, whose operators `-` and `*` are overloaded to generate an invocation sequence. The Scala compiler checks if all functions forming an invocation sequence share the same signature. An invocation sequence sets its terminating condition by calling the `terminate` method, and executes the equivalent functions by calling the overridden `apply` method, returning a mapping of `(functionName, executeResult)`.

We observe that an invocation sequence naturally maps into a tree structure that can serve as its runtime representation. The tree structure's nodes have three types: `leaf`, `sequential`, and `parallel`. A `leaf` is an equivalent functionality. A `sequential` node has its `left` and `right` children, and a `parallel` node has two or more `child` nodes.

Algorithm 2 explains how to implement the runtime using the tree structure. To create and manage concurrency, our implementation uses the `Future`, `Promise`, and `concurrentMap` APIs. A concurrent access protected key-value data structure (`resultMap`) maps the completed equivalent functions and their results. A recursive procedure starts from the tree structure's

Algorithm 2 Execute a Specified Meta-pattern

```
Input: p: execution parameter; < $\theta$, $\zeta$, $\omega$ >
Output: r: result
1: execute($\zeta$.root)
2: function EXECUTE(t:TREE)(Boolean)
3:     switch t.Type do
4:         case Leaf(v)
5:             resultMap ← resultMap + (v.funcName, v.func(p))
6:             return terminator.check(resultMap)
7:         case SequentialNode(left, right)
8:             if execute(left) then
9:                 return true
10:            else
11:                return execute(right)
12:            end if
13:        case ParallelNode(chidren)
14:            $fSet \leftarrow \emptyset$
15:            for each $c \in$ children do
16:                $fSet \leftarrow fSet$ + Future(execute(c))
17:            end for
18:            Wait any $f \in fSet$.Complete:
19:            if f==True then
20:                return True                                        ▷ Early Termination
21:            else if fSet.all.isCompleted then
22:                return False
23:            else
24:                continue Wait
25:            end if
26: end function
```

root node, and returns true, as soon as the terminating condition is fulfilled. Upon reaching a leaf node, its equivalent function is executed, with the result stored in the key-value structure. All the stored results are checked after each completed function if the pattern's terminating condition has been fulfilled (line 6). For a sequential node, its left node is executed first, followed by executing its right node if the recursive procedure of its left node returns false. For a parallel node, all child nodes are executed in parallel, and the parallel node waits for the results of these recursive procedures. If any of the child nodes fulfills the terminating condition and returns true, the parallel node returns true (line 20) and the parallel execution is terminated without waiting for the other branches to complete. Otherwise, it continues to wait, until all child nodes' executions fail to fulfill the terminating condition and return false. After the recursive procedure completes, it returns the stored final results to the caller.

YAML: To demonstrate how our meta-pattern can be integrated into the state of the practice workflow orchestration languages, we apply the BPEL design to implement a YAML-based procedural language for declaratively specifying patterns. A YAML parser in Scala parses YAML scripts to be further processed by our Scala library.

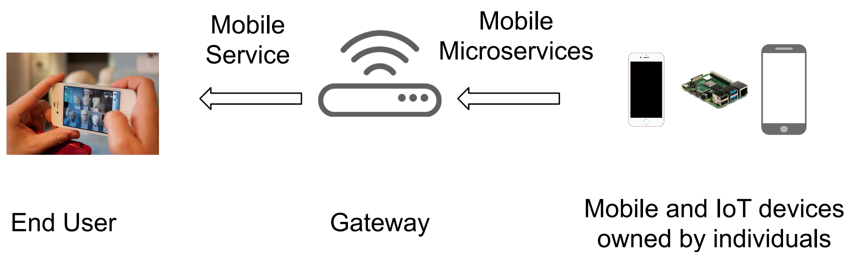


Figure 5.17: System Components for Provisioning Mobile Services

Applying Meta-Pattern to Mobile Service We adopt the mobile services provisioning system model introduced in [151]. In particular, the system features a local gateway that collects the available microservices, provided by mobile and IoT devices. For a given mobile service request with reliability, trustworthiness, and QoS-optimality requirements, the gateway orchestrates the combined execution of equivalent microservices, provided by mobile and IoT devices, which can be unreliable and untrustworthy.

Enhancing Service's Accuracy: To detect faces, developers can choose proprietary cloud services (IBM[1], Microsoft[2], and Face++[3]) or deploy open-source libraries as edge services (deep learning based[4] and openCV Cascade classifier based[5]). Fig. 5.18 shows how with our Scala library, a fine-grained workflow pattern that enhances service accuracy can be implemented in 4 lines of code. Lines 1-11 implement microservices `ibm, ms, face, dl , opencv`, taking a String (i.e., image file) and returning a Boolean (i.e., face detected).

[1]https://www.ibm.com/watson/services/visual-recognition/
[2]https://azure.microsoft.com/en-us/services/cognitive-services/face/
[3]https://www.faceplusplus.com/face-detection/
[4]https://github.com/ageitgey/face_recognition
[5]https://docs.opencv.org/3.4/d1/de5/classcv_1_1CascadeClassifier.html

```
 1 // invoke a web service:
 2 def ibm(image:String):Boolean = {...}
 3 def ms(image:String):Boolean = {...}
 4 def face(image:String):Boolean = {...}
 5 // invoke an edge service:
 6 def dl(image:String):Boolean = {
 7    val reg = new EdgeReg() //connect to an edge gateway
 8    val edgeService = reg.query("deepLearningFaceDetection")
 9    result = edgeService.execute(image)
10 }
11 def opencv(image:String):Boolean = {...}
12 // Specify equivalent microservices
13 val (e1, e2, e3, e4, e5) = (eqv(ibm), eqv(ms), eqv(face), eqv(dl), eqv(opencv))
14 // Specify an invocation sequence
15 val seq = e4*e5 - e1*e2*e3
16 // Specify a terminating condition
17 seq.terminate(majorityVoting())
18 // Execute and process result
19 val result = seq('img.jpg').groupBy(_._2).maxBy(_._2.size)._1
```

Figure 5.18: Specifying Mobile Service in Scala Library

Line 13 wraps them up in equivalent microservice containers. Line 15 uses the overloaded operators to declare an invocation sequence, and Line 17 sets the terminating condition for the invocation sequence. Line 19 executes the specified pattern with the input of "`img.jpg`" and obtains the execution result that is agreed by most microservices.

Enhancing Service's Reliability: Edge environments typically feature multiple sensors, whose input can be used to satisfy the data requirements of an edge service. Consider a service that obtains ambient temperature. Ambient temperature can be obtained by: 1) directly reading a local temperature sensor (`readTempSensor`); 2) estimating based on the CPU temperature of an edge computer [84] (`estTemp`); 3) reading from a web service based on the current location (`readLocationTemp`). In most edge environments, the `readTempSensor` microservice is first executed to provide a low-latency, low-cost, and accurate temperature reading. However, if some microservices are unavailable in a given environment, `estTemp` and `readLocationTemp` are executed next in parallel as fail-over to guarantee an acceptable latency. Fig. 5.19 shows a YAML script file that expresses the workflow pattern for implementing service `getTemp`. In particular, the invocation sequence is specified in Line 2, and the terminating condition is specified in line 3.

```
1 id: getTemp
2 executionSeq: e1 - e2*e3
3 terminate: anyResult
4 microservices:
5 - {id: e1, microservice:readTempSensor}
6 - {id: e2, microservice: estTemp}
7 - {id: e3, microservice: getLocationTemp}
```

Figure 5.19: Specifying Mobile Service in YAML Script

Performance Evaluation Enhancing Accuracy:

id	func	cost	latency (ms)	accuracy
e1	IBM	400	95	0.918
e2	MS	100	74	0.737
e3	Face++	50	96	0.898
e4	DL-based	2	56	0.642
e5	openCV-based	2	66	0.676

Table 5.6: QoS of Facial Detection Microservices

To verify how the generated workflow patterns work for the aforementioned application scenario, we use the image dataset collected from WiKi [74] as an alternative for human labeling, in which each image contains a face. We deploy the edge services on a Dell desktop with a i7-4790@3.6GHz processor and 16GB RAM. Table 5.6 shows the average latency, accuracy, and cost of each equivalent microservice. The cost of invoking the web services provided by IBM, Microsoft, and Face++ are \$0.004, \$0.001, and \$0.0005 per request, respectively. Assuming the average electricity rate of \$0.12 per kWh, and the power supply of the experimental desktop of 0.65kW, the costs of microservices become 400, 100, 50, 2, and 2, respectively.

Table 5.7 compares the QoS of crude-grained majority voting and fine-grained patterns generated by our meta-pattern. We observe that: 1) the generated plan can be extremely cost/latency efficient by first executing the two open source implementations deployed at the

invocation sequence	accuracy	cost	latency
Crude-grained Majority Voting	0.859	554	97
$e_4 * e_5 - e_2$	0.739	23	81
$e_4 * e_5 - e_3$	0.812	13	85
$e_1 * e_3 * e_5$	0.908	452	95
$e_3 * e_5 - e_1$	0.908	110	110
$e_2 * e_3 - e_1$	0.939	311	134

Table 5.7: QoS of Facial Detection Services

edge. Compared with invoking the IBM web service, the generated pattern saves as much as 97% of the execution cost, while reducing the accuracy by 13%; 2) the workflow pattern $e_3 * e_5 - e_1$ strikes a good balance between accuracy and cost. By invoking a low-cost web service and an open-source implementation first, the execution gains more accuracy than when using the two cost-efficient open source implementations. Compared with invoking the IBM service, the generated pattern saves 72.5% of execution cost, while the accuracy only decreases by 3.2%; 3) compared with the crude-grained majority voting pattern, the generated patterns on average reduce the cost by 67.6%, with less than 3% differences in accuracy and latency. This observation confirms our motivation: fine-grained workflow patterns do optimize performance.

Pattern	Invocation Sequence	Reliability	Latency	Cost
Speculative Parallel	$e_1 * e_2 * e_3$	100%	56 ms	148
Fine-Grained	$e_1 - e_2 * e_3$	99%	69 ms	74.5

Table 5.8: QoS of Service "getTemp"

Enhancing Reliability: The execution environment features a mobile device (Moto G6) that queries the "getLocationTemp" microservice, a temperature sensor (Raspberry Pi 3 and DS18B20) for "readTempSensor", and an edge server (ThinkCentre M900 Tiny) for "estTemp". The sensor is only available for 60% of all requests. We further set the cost for executing each microservice to 50 points. Table 5.8 compares the reliability, latency, and cost of the speculative parallel execution and the fine-grained workflow pattern. The fine-grained pattern reduces the average latency by 49.7%, at the expense of 23.2% additional latency, as compared with the speculative parallel workflow pattern.

5.2.5 Conclusion

We have presented a meta-pattern that declaratively expresses fine-grained workflow patterns for the combined execution of equivalent microservices to improve QoS. The meta-pattern declaratively specifies a fine-grained pattern as a set of equivalent microservices, an invocation sequence, and a terminating condition. Our evaluation demonstrates that our approach is expressive and effective, presenting a viable solution that helps conquer the complexity of reliable, accurate, and efficient execution in distributed execution environments with scarce and unreliable resources.

Chapter 6

Adaptive Edge Services

Edge computing coordinates sensing, computation, and data storage resources at the edge of the network [148]. Being within the direct communication range of each other and the client, resource-providing edge devices offer the communication latency lower than that of cloud-based servers [47]. One way to expose edge-based resources to application programmers is via the service-oriented architecture (SOA). A service coordinates the execution of edge-based distributed tasks, implemented as edge microservices [153].

When it comes to provisioning services, cloud-based systems coordinate the execution of abundant and reliable resources. In contrast, *edge-based systems coordinate the execution of unreliable and dynamic resources.* The execution failure ratio of edge services tends to be higher than that of cloud services [9, 87], as it is often mobile or energy-harvesting [49, 111, 180] devices that supply edge resources. In edge environments, an execution can fail for multiple reasons: a mobile device moves out of communication range; an energy harvesting device becomes temporally unavailable, driven into sleep mode; a speech recognizer fails due to noise. Besides, cloud service vendors can always cost-efficiently allocate the required amount of pre-deployed resources, while edge services may need to be provided in diverse edge environments with dissimilar and often scarce resources [17].

To improve reliability, the state of the practice for cloud systems is to deploy replicated services on redundant cloud resources. On the contrary, edge systems rely on resource-scarce edge devices, rendering the replication solution inapplicable. Considering the wide range of sensors and data processing methods at the edge, our previous work MOLE [153] takes advantage of equivalent microservices, which provide the same functionality by different means and rely on dissimilar resources (e.g., (1) camera/image analysis, (2) motion sensors, and (3) wireless signal, used in place of each other for indoor localization [26]). These equivalent microservices can be executed in the fail-over pattern to improve reliability with minimal costs or in the speculative parallel pattern to improve reliability with minimal latency; we call such patterns *execution strategies*.

However, MOLE cannot always deliver QoS-consistent edge services, as it follows the specified fixed execution strategy across edge environments with vastly dissimilar resources. The resource dissimilarity across different environments yields constituent equivalent microservices with uncertain QoS, which in turn results in edge services that execute these microservice in predefined patterns delivering unpredictable and inconsistent QoS to the client. The state of the art lacks a frame of reference for identifying and expressing highly customized strategies for executing equivalent microservices, whose QoS performance can be estimated accurately.

In the approach presented herein, we provide reliable and QoS-consistent edge services with unreliable and dynamic resources. In particular, rather than follow predefined execution strategies (as in MOLE), we provide highly customized execution strategies that increase the QoS-consistency of edge services across edge environments. Our system employs a feedback loop [18, 32] to monitor the environment-specific performance of edge microservices and dynamically generate execution strategies based on the service's QoS requirements.

The insight that motivates our system design is the dissimilar QoS of executing equivalent microservices by different strategies. To be able to generate a customized execution strategy that best fits the QoS requirements in a given edge environment, we explore the following system design questions: 1) how to express customized execution strategies; 2) how to determine all possible strategies for a given set of equivalent microservices; and 3) how to estimate the QoS of a strategy.

The contribution of this work is threefold:

- **System Design**: We introduce an edge system design that provides reliable services with consistent QoS. Our design features a feedback loop that collects the environment-specific performance of microservices, as well as a generator that customizes execution strategies to best satisfy services' QoS requirements. To the best of our knowledge, this paper is the first to identify, define, and solve the problem of providing QoS-consistent services in dissimilar edge environments with dynamic resources.
- **Customized Execution Strategies**: We explore how to customize execution strategies of equivalent functionalities to best satisfy given QoS requirements. To the best of our knowledge, we are the first to be able to 1) formulate any customized execution strategy for equivalent functionalities; 2) determine what all possible execution strategies for any number of equivalent functionalities are and estimate their QoS.
- **Evaluation**: We systematically evaluate the efficiency and scalability of our system design as well as its actual performance by benchmarking edge services deployed and executed in real execution environments.

6.1 Problems in Provisioning Edge Services

By embracing the service-oriented architecture (SOA), edge executions across heterogeneous distributed devices are exposed as service invocations, thus shielding application developers from the necessity to implement low-level, platform-specific functionalities and D2D communication. Although SOA has become an industry standard for cloud computing [163], edge computing operates in fundamentally different execution environments, rendering cloud-based SOA designs inapplicable. While to meet the service level agreements for cloud services, their vendors only need to appropriately configure the abundant computational and network resources, edge service providers often have scarce, unreliable, and dynamic resources at their disposal, with which to meet the QoS requirements. To demonstrate the problems that these realities of edge computing present, consider the following example.

6.1.1 Motivating Example: Detecting Fire

One of the key functionalities of personal mobile assistants is to keep their owners safe. Such assistants can have a feature that periodically checks for the potential presence of fire to be able to alert its users and guide them to an escape route. To detect the presence of fire in the surrounding environment, the edge service `detectFire` can be queried in dissimilar environments that can range from office buildings to apartments, shopping malls, and even campgrounds. This service must be reliable, responsive, and cost efficient.

What hinders the QoS-guaranteed provisioning of such a service in dissimilar edge environments is their unreliable and dynamic resources [8, 21, 22, 87, 133, 153]. Fig. 6.1 demonstrates how a mobile device queries edge services in edge environments with dissimilar resources:

1) Edge resources can be provided by mobile devices [10, 118] or energy harvesting stationary devices [111, 180]. With multiple mobile devices in the vicinity, they can be organized into a computing ensemble that can execute demanding edge services [10, 118]. However, typically owned by individuals, mobile devices are hard to predict or control, as their owners can move away or use them at any time, thus causing service failures. Besides, IoT devices increasingly rely on the energy harvesting technology [75], which accumulates ambient recyclable energy,

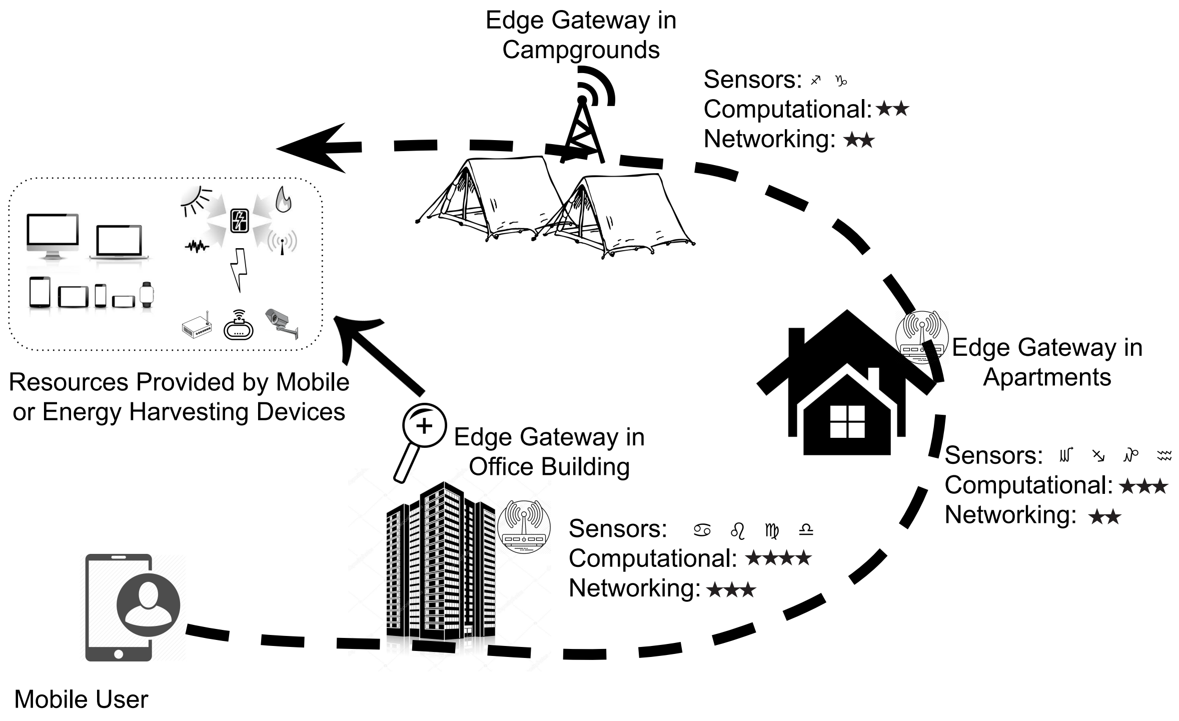


Figure 6.1: Edge Services in Dissimilar Edge Environments

including solar radiation, wind, human motion energy, and WiFi signals. However, these devices can be operated only intermittently: energy may be unavailable to harvest, taking time to accumulate to allow execution [104]. As a result, when executed on such devices, microservices cannot guarantee satisfactory reliability.

2) Besides, different edge execution environments may possess resources with dissimilar capabilities and capacities. For example, an office building may have built-in flame sensors for detecting fire, while apartments may only have smoke detectors; an indoor environment may have high-performance edge servers for computationally intensive tasks, while an outdoor environment may only have a solar-powered Raspberry Pi with much lower computational power. The resource difference across edge environments causes the dissimilar availability and performance of edge-based microservices.

6.1.2 MOLE: Reliability-enhanced Edge Services

Our previous work MOLE [153] presented in Chapter 4, demonstrates that equivalent functionalities can be executed to improve the reliability of edge computing. Edge computing environments feature a wide range of sensors and data processing methods, so an application requirement can be fulfilled in multiple equivalent ways. MOLE enables edge service developers to specify the execution strategies for equivalent microservices, which include the fail-over and speculative parallel strategies. The `fail-over` strategy first tries executing a microservice; if it is unavailable or disabled, the execution switches to a back-up microservice. The `speculative parallel` execution strategy spawns the execution of all microservices simultaneously, proceeding as soon as any of them returns successfully. With both strategies improving reliability, fail-over is cost-efficient and speculative parallel is latency-efficient.

In the aforementioned example, to improve its reliability, `detectFire` can execute the equivalent microservices that detect 1) smoke by a surveillance camera; 2) smoke by smoke sensors; 3) flame by flame sensors; 4) the change of CO/CO_2 level by gas sensors; 5) the temperature change by a temperature sensor. We assume that the output of any one of these microservices, rather than their fusion, can detect fire. When one microservice fails, MOLE switches to its equivalent pair. Even if one or more microservices are unreliable, the edge service's overall reliability can still be guaranteed.

However, the overall performance of MOLE-specified services differs across edge environments with predefined execution strategies. For example, assume the `detectFire` service is developed in an environment `A` with edge-based small-scale data centers providing the computational power. Considering the latency of each equivalent microservice is pretty low, the developer specifies the execution strategy as "fail-over" for better cost-efficiency. However, while being executed in a different edge environment `B` with a Raspberry Pi providing the computational power, the “fail-over” execution may lead to an extremely long latency which is unexpected. Hence, our solution extends MOLE’s reliability enhancement by introducing a novel system design that uses a feedback loop to generate environment-tailored execution strategies.

6.1.3 Customizing Execution Strategies to Optimize QoS

Due to the proliferation of unreliable execution environments (e.g., edge, IoT, etc.), the problem of optimizing their QoS has come to the forefront of distributed system design. This problem is exacerbated by these environments being unable to take advantage of existing designs that rely on standard resource deployments. In the approach presented herein, we put forward a novel optimization methodology that customizes the execution strategies for equivalent microservices.

Several prior approaches make use of the combined execution of equivalent functionalities. To improve service responsiveness, several cloud service instances are deployed and executed simultaneously [50, 132]. To improve reliability, automatic Workarounds provide automatic fail over with equivalent functionalities [23]. The emergence of IoT and edge computing gives rise to distributed execution environments that feature a wide range of sensors and processing methods, thus greatly increasing the variety and number of equivalent functionalities. However, all these existing approaches can execute equivalent functionalities in predefined execution patterns. The state of the art lacks a frame of reference for identifying and expressing highly customized strategies for executing equivalent microservices, whose QoS performance can be estimated accurately. The exploding numbers of equivalent functionalities of the emerging distributed environments present an untapped potential for optimization QoS by fully exploiting their customized execution.

6.2 Execution Strategies for Equivalent Microservices

Consider the aforementioned example: we use a, b, c, d, e to denote the five equivalent microservices for `detectFire`. For example, possible execution strategies for five equivalent microservices (a, b, c, d, e) include, but are not limited to: 1) fail-over: execute a, b, c, d, e in turns if the previous microservice fails; 2) speculative parallel: execute a, b, c, d, e simultaneously, returning the first obtained result; 3) first execute a and b simultaneously; if any of them succeeds, return the results; otherwise, execute c, d, e simultaneously and return the first available result; 4) first execute a, then b and c simultaneously; if none of them succeed, execute d first then e. To generate execution strategies that best satisfy given QoS requirements, we need to 1) find all possible execution strategies and 2) compare their QoS.

6.2.1 Determining all Possible Strategies

The problem we solve in this subsection is, *given a number of equivalent microservices (i.e., 3 microservices a, b, and c), how to find all possible strategies to execute them?*

Our solution is inspired by the exhaustive search solution for the 24 game, which is a classic math game: given 4 numbers in the range from 1 to 9, binary operators (+, -, *, /), and parentheses, form an arithmetic expression that equals to 24. The exhaustive search solution [177] lists all possible expressions and removes duplicates. To generate all expressions, the solution proceeds in three steps: 1) put all digits into 4 slots, resulting in P(9, 4) arrangements; 2) for each arrangement, put any one of the 4 operators into each of the 3 slots between the digits, resulting in (P(9, 4)*4^3) arrangements; 3) to process parentheses, alter the precedence of the 3 operator slots. The number of final expressions is P(9, 4)*4^3*P(3,3).

We convert the problem of "finding all possible execution plans for an equivalent set of size n" to "finding all execution plan equations that contain m ($1 \leq m \leq n$) equivalent functionalities out of n, with $m-1$ operators out of $-$ and $*$, and parentheses." We first apply the aforementioned exhaustive search to find $P(n,m) * 2^{m-1} * (m-1)!$ expressions of execution plans for all m $\in [1, n]$, remove the duplicate expressions, and put them together to produce the answer. The duplication removal procedure takes advantage of the three observations above to identify duplicate expressions.

For each $2 \leq n \leq 6$, Table 6.1 gives the number of distinct execution strategies for an equivalent set of size n. $F(M)$ denotes the size of strategies that contains all M microservices, while $F'(M)$ denotes the size of the strategies that contains 1 to $|M|$ microservices. We observe that as few as four equivalent functions can have over 200 possible execution strategies.

M	2	3	4	5	6
$F(M)$ (with M microservices)	3	19	207	3211	64743
$F'(M)$ ($\forall n \in [1, M]$ ms)	5	31	305	4471	87545

Table 6.1: Execution Strategies for M Eqv MS

Fig. 6.2 lists all possible strategies (separated by ";") to execute all three equivalent microservices a, b, and c. Given the QoS of these microservices, executing all of them following these different strategies may lead to 19 dissimilar values of overall QoS in an edge environment.

```
1.a - b - c; 2.a - c - b; 3.b - a - c;
4.b - c - a; 5.c - a - b; 6.c - b - a;
7.a * b - c; 8.a * (b - c); 9.a * c - b;
10.a * (c - b); 11. a - b * c; 12. a * b * c;
13.b * (a - c); 14.b * c - a;
15.b * (c - a); 16.b - a * c;
17.c * (a - b); 18.c * (b - a); 19.c - a * b;
```

Figure 6.2: Execution Strategies for Eqv MS (a, b, and c)

6.2.2 Estimating the QoS of a Strategy

This subsection presents a solution to the following problem: *given the QoS of a, b, c, what is the average QoS of executing $a * b * c$ multiple times?* Some existing solutions estimate the overall QoS of an execution strategy by folding the QoS calculation over a collection of equivalent services [57]. We will compare our results with theirs. We are not estimating the QoS of one execution, as any microservice could fail or succeed, leading to dissimilar performance. Instead, we estimate the **average** QoS of running an execution strategy multiple times.

QoS Model and Assumptions

We consider three major QoS attributes for edge services and microservices: **cost**, **latency** (or say, response time), and **reliability**. The cost attribute is estimated as the amount of energy consumed to execute edge microservices. The latency attribute refers to the time taken to execute microservices and services. The reliability attribute refers to the probability of finishing an execution successfully. As the execution status in edge environments differs across runs, we compute the QoS as the average value of multiple executions in an edge environment.

$\mathcal{M} = \{m = 1, 2, ...M\}$ denotes a set of equivalent microservices, while r_m, l_m, c_m denote the average reliability, latency, and cost of $\forall m \in \mathcal{M}$ in an edge environment. Our QoS estimation is based on the following assumptions:

Assumption 1: although multiple devices can provide a microservice in an edge environment, our system only selects the one with the best QoS;

Assumption 2: once an edge device receives a microservice execution request, it charges the microservice's full execution cost, irrespective of whether the execution is to succeed, fail, or terminate midway.

QoS Estimation Algorithm

We estimate the QoS of a strategy as follows. The overall reliability of a strategy can be directly estimated as $r = 1 - \prod_{m}^{m \in \mathcal{M}} (1 - r_m)$, as a strategy only fails when all its constituent microservices fail. For cost and latency, we first convert the expression of an execution strategy to a tree structure, which has three node types: `leaf`, `sequential`, and `parallel`. A `leaf` is an equivalent microservice. A `sequential` node has its `left` and `right` children, and a `parallel` node has two or more `child` nodes.

Algorithm 3 shows how to estimate the cost and latency using a tree. Starting from the `root` of the tree, it recursively calculates the timelines for all microservices (Lines 15 to 33). A timeline ($\tau = (m, s, e)$) denotes a microservice m, its start time s and end time e. For a `leaf` node, m points to its microservice, with start time set to 0 ($s = 0$) and end time set to the latency of the microservice ($e = l_m$). For a `sequential` node, the time lines of its `left` and `right` children are generated. The longest end time of microservices belonging to the `left` child is added to the start time and end time of each microservice belonging to the `right` child (Lines 23 to 25), as the right child of a sequential node is only executed when all microservices in the `left` child fail. For a `parallel` node, the timelines of all its children are generated.

Lines 3 to 7 calculate the latency of a strategy. The timelines are sorted by their end time in ascending order to form a list ϕ. The overall latency is calculated as follows: for each microservice ($\phi(i)$), add up its end time multiplied by the probability that the overall execution terminates upon the microservice completing its execution (the probability that

Algorithm 3 Estimate Cost, Latency for a Strategy

```
Input: $es$: strategy
Output: $l$: latency, $c$: cost
1: $l \leftarrow 0, c \leftarrow 0$
2: $\tau \leftarrow$ GetTimelines($es$.root)
3: $\phi \leftarrow \tau$.sortBy(e)                                    ▷ sort by endTime
4: for  $i \leftarrow 0$ to $|\phi| - 2$  do
5:     $l+ = \left(\prod_{j=0}^{i} (1 - r_{\phi(j).m})\right) * r_{\phi(i).m} * \phi(i).e$
6: end for
7: $l+ = \left(\prod_{i=0}^{|\phi|-2} (1 - r_{\phi(i).m})\right) * \phi(|\phi| - 1).e$
8:
9: for  $(m, s, e) \in \tau$ do
10:     $\xi \leftarrow \tau$.filter(_.$e < s$)                     ▷ any ms finishs before $m$ starts
11:     $c+ = \prod_{j=0}^{|\xi|-1} (1 - r_{\xi(j).m}) * c_m$
12: end for
13: return $l, c$
14:
15: function GetTimelines($t$:Tree)($\{\tau = (m, s, e)\}$)
16:     switch $t$.Type do
17:         case Leaf
18:             return $\{(t$.func, 0, $t$.func.latency$)\}$
19:         case SequentialNode
20:             $\tau_l \leftarrow$ GetTimelines($t$.left)
21:             $t_{left} \leftarrow \max(\tau_l.e)$
22:             $\tau_r \leftarrow$ GetTimelines($t$.right)
23:             for $i \in \tau_r$ do
24:                 $i.e \leftarrow i.e + t_{left}$, $i.s \leftarrow i.s + t_{left}$
25:             end for
26:             return $\tau_l \cup \tau_r$
27:         case ParallelNode
28:             $\tau \leftarrow \emptyset$
29:             for $i \in t$.children do
30:                 $\tau \leftarrow \tau \cup$ GetTimelines($i$)
31:             end for
32:             return $\tau$
33: end function
```

all microservices in front of $\phi(i)$ fail and $\phi(i)$ succeeds). Lines 9 to 12 calculate the cost of a strategy. The overall cost is calculated as follows: for each microservice m, add up its c_m multiplied by the probability that the overall execution would not terminate before it has a chance to execute (i.e., all microservices in ξ fail, with ξ denoting all microservices that finish before m starts).

QoS Estimation Example

For example, consider the $a * b * c$ strategy, in which $l_a = 10ms$, $r_a = 10\%$, $l_b = 90ms$, $r_b = 90\%$, $l_c = 70ms$, and $r_c = 70\%$. By using our QoS estimation method, the latency of the aforementioned $a * b * c$ would be estimated as: $10 * 10\% + 70 * (1 - 10\%) * 70\% + 90 * (1 - 10\%)(1 - 70\%) = 69.4ms$

The folding based method [57] has also been applied to estimate the QoS of a strategy. It first calculates the latency and reliability attributes of $\theta = a*b$ as: $l_\theta = 10*10\%+90*(1-10\%) = 82ms$, $r_\theta = 1-(1-10\%)*(1-90\%)$=91%. Then it computes $\theta * c$, leading to an estimated overall latency of $70 * 70\% + 82 * (1 - 70\%) = 73.6ms$. However, this estimation fails to consider how the execution status of services that appear later on the list affect the execution of services preceding them. If, for example, c successfully completes its execution first, its result will be used right away, without waiting for b to complete its execution. Our evaluation in Section 6.4 confirms the correctness of our method.

6.2.3 Execution Strategy Examples

For the aforementioned fire detection example, we set the QoS, $[cost, latency, reliability]$ of microservices $a - e$ to $[50, 50, 60\%]$, $[100, 100, 60\%]$, $[150, 150, 70\%]$, $[200, 200, 70\%]$, and $[250, 250, 80\%]$. Table 6.2 lists example strategies and their resulting QoS, calculated by the methodology introduced in this section. We observe that compared with the predefined strategies (strategies 1 & 2), the customized strategies (strategies 3 & 4) strike better balance between the QoS attribute values. For example, if latency is the major concern, strategy 2 is the most latency-efficient but cost-inefficient, while strategy 4 reduces the cost by 50.6% with a minor increase on the latency (5%). This example demonstrates how executing equivalent microservices by different strategies leads to vastly dissimilar QoS.

id	Execution Strategy	cost	latency	reliability
1	a-b-c-d-e	126	126	99.7%
2	a*b*c*d*e	750	81	99.7%
3	a-b*c-d-e	162	111	99.7%
4	c*(a*b-d*e)	372	85	99.7%

Table 6.2: Execution Strategies and Estimated QoS

6.3 System Design and Strategy Generation

The design of our edge-based service provisioning system follows and extends that of MOLE [153]. In particular, we extend the edge gateway to support new workflows to provision QoS-consistent edge services.

6.3.1 System Components and Edge Service Execution

Fig 6.3 shows the main components and service provisioning workflow of our design, which features a client, an edge gateway, multiple edge devices for executing microservices, and a cloud-based market that hosts self-describing scripts for services and microservice executables.

A client device sends edge service requests, identified by a unique `ServiceID`, to its connected gateway. The gateway follows a service script describing the dataflow of constituent microser-

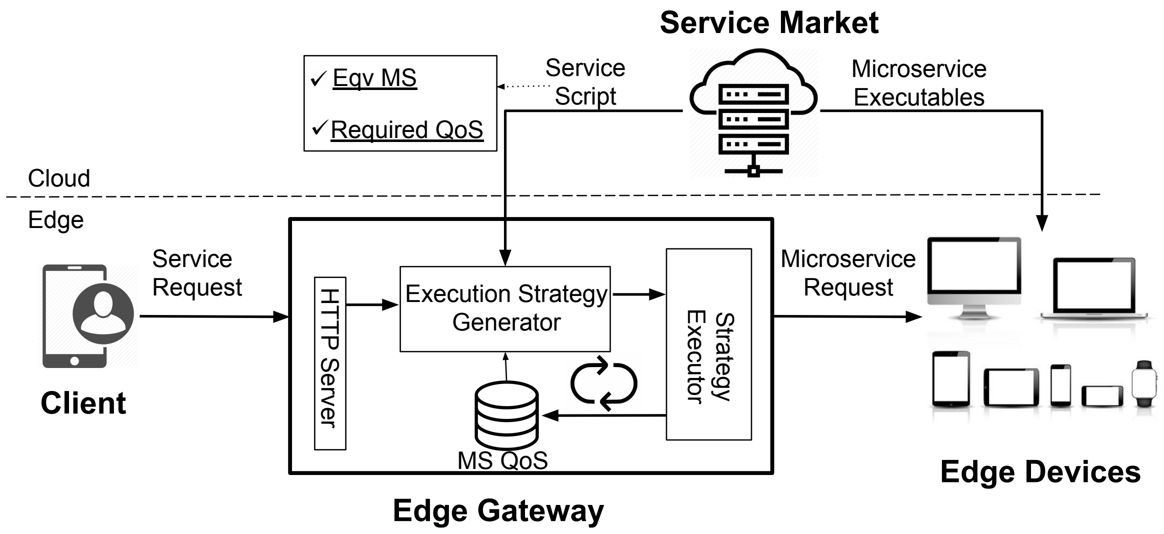


Figure 6.3: System Design for Provisioning Edge Services

vices and the QoS requirements to invoke microservices that are further being executed on edge devices. The service scripts required by the gateway and microservice executables required by the edge devices can be downloaded from a service market, and cached locally for further executions. Hence, if a recently executed service is invoked again, the request can be processed entirely within the edge's local environment, without needing to interact with the cloud.

The runtime starts executing a service by following the default execution strategy to collect the environment-specific non-functional performance attributes for each invoked microservice. As the service continues being invoked, a generator (on the gateway) synthesizes an execution strategy that satisfies the QoS requirements more closely by adapting to the changed performance of the constituent microservices. That strategy executes until a successor with better QoS replaces it, so the system self-adapts to dissimilar edge environments.

6.3.2 Major Enhancements Over MOLE

In MOLE, a service script specifies a prioritized list of equivalent microservices. A script is then uploaded to a cloud-based service market to be transformed into an execution strategy, based on the priorities of the constituent equivalent microservices and the developer specified execution strategies. Different from MOLE, our system generates the execution strategies locally at the edge gateway, to accommodate the edge-specific performance of the microservices.

In addition, the edge gateway now involves a feedback loop that comprises an execution strategy generator, a collector for recording microservice QoS characteristics, and a strategy executor. Upon receiving a service request, the gateway imports the corresponding service script, reading the QoS of microservices and the service's QoS requirements. An execution strategy generator retrieves the QoS of constituent microservices from the collector, and outputs an execution strategy. The strategy executor follows the strategy to invoke microservices. The collector keeps updating the QoS characteristics of microservices until their executions complete.

6.3.3 QoS Utility Index

The QoS satisfaction model for cloud services is binary: given a set of QoS requirements and a service's SLA, the service either satisfies the requirements or not. Application developers select to integrate only those services that satisfy the QoS requirements. However, with edge applications, developers may have no alternatives and can only use the available edge services, rendering the binary QoS satisfaction model inapplicable. Although QoS requirements are still imposed, applications may need to integrate with edge services that approximate the requirements most closely, and that is what our strategy generation aims for. If a generated strategy fails to reach one or multiple required QoS attributes specified in service scripts, the gateway reports the estimated unsatisfied QoS to the client, which then determines whether the service request with this expected QoS should be continued.

QoS has multiple attributes. For generality, we consider $\mathcal{N}$ QoS attributes, with $n = |\mathcal{N}|$. For example, in our system model, $\mathcal{N} = \{c, l, r\}$, so $n = 3$. $\mathcal{Q}_n$ denotes the requirement of QoS attribute n imposed on an edge service. In our system model, Q_r, Q_c, Q_l denote the requirements on reliability, cost, and latency, respectively. $S = \{s = 1, 2, ..., |F(M)|\}$ denotes all possible strategies, while $Q(s) = \{q_1(s), q_2(s), ..., q_n(s)\}$ denotes the estimated QoS of strategy s. QoS attributes can be placed in the following two categories, as based on their optimization criteria: 1) the smaller the better, denoted as $\mathcal{N}_-$ (i.e., cost and latency); 2) the higher the better, denoted as $\mathcal{N}_+$ (i.e., reliability and trust level). For any QoS attribute $n \in \mathcal{N}$, $q_n \preceq q'_n$ denotes q_n is worse than or equals to q'_n (i.e., $q_n \le q'_n$ if $n \in \mathcal{N}_+$, or $q'_n \le q_n$ if $n \in \mathcal{N}_-$), and $q_n \succ q'_n$ denotes q_n is better than q'_n.

Among S, the QoS of a subset of strategies are Pareto optimal[113]. A strategy s is Pareto optimal `iff` no other strategies in $\mathcal{S}$ can improve any of the QoS attributes without worsening the remaining QoS attributes $\big($i.e., $\nexists s' \in \mathcal{S}$, that satisfies: $\forall n \in \mathcal{N}, q_n(s) \preceq q_n(s')$ and $\exists n \in \mathcal{N}, q_n(s') \succ q_n(s)\big)$. To evaluate how these Pareto optimal strategies satisfy the QoS requirements, we introduce a utility index $U(s) = \sum_n^{\mathcal{N}} u_n(s)$, where

$$u_n(s) = \begin{cases} -k\dfrac{|q_n(s) - Q_n|}{Q_n}, \text{if } q_n(s) \preceq Q_n \\ \dfrac{|q_n(s) - Q_n|}{Q_n}, \text{if } q_n(s) \succ Q_n \end{cases} \forall n \in \mathcal{N}, k > 1 \tag{6.1}$$

In the equation above, $\frac{|q_n(s)-Q_n|}{Q_n}$ denotes the normalized distance between a strategy's estimated value and the requirement imposed on the QoS attribute n. $u_n(s)$ is positive when $q_n(s) \succ Q_n$, negative when $q_n(s) \prec Q_n$, and zero otherwise. However, when the requirement is not satisfied (i.e., $q_n(s) \succ Q_n$), $u_n(s)$ changes at a higher rate due to the system parameter k. The reasoning behind this index is that even for a fully satisfied QoS attribute, its improvement can still increase the overall utility; however, the rate of the increase would be slower than when the QoS attribute is unsatisfied.

To demonstrate how the utility index metric works, consider two strategies s_1 and s_2. s_1 delivers exactly the required cost, latency, and reliability, while s_2 improves cost and reliability by 5% each at the expense of 10% additional latency. With k as a penalty for unsatisfied QoS attributes, the utility of s_1 is higher than that of s_2. A higher k value can be specified to incur a higher penalty for unsatisfied QoS attributes. For example, assume s_2 improves cost and reliability by 10% each at the expense of 10% additional latency; hence, $u(s_1) = u(s_2) = 0$ if $k = 2$, while $u(s_1) = 0 > u(s_2) = -0.1$ if $k = 3$.

Algorithm 4 Strategy Generation

```
Input: $\mathcal{M}$: equivalent microservices
Output: $es$: execution strategy
 1: if $|\mathcal{M}| > \theta$ then
 2:     $es \leftarrow \text{exhaustiveSearch}\big(\text{strategies}(|\mathcal{M}|)\big)$
 3: else
 4:     $\mathcal{M}' \leftarrow sortByUtility(\mathcal{M})$
 5:     $es \leftarrow \mathcal{M}'(0)$
 6:     for $i \leftarrow 1$ to $|\mathcal{M}'| - 1$ do
 7:         $es_1 \leftarrow es - \mathcal{M}'(i)$ , $es_2 \leftarrow (es) * \mathcal{M}'(i)$
 8:         if $\text{utility}\big(es_1\big) > \text{utility}\big(es_2\big)$ then
 9:             $es \leftarrow es_1$
10:         else
11:             $es \leftarrow es_2$
12:         end if
13:     end for
14: end if
15: return $es$
```

6.3.4 Generation Heuristic

The pseudo code in Alg. 4 shows our strategy generation heuristic. To generate execution strategies time-efficiently, we use the exhaustive search when the number of equivalent microservices is small, and switch to an approximation heuristic when the number exceeds a threshold causing the exhaustive search to take too long to finish. For a set of M equivalent microservices, the exhaustive search estimates the QoS performance for each possible execution strategy that contains all M microservices (i.e., $F(M)$), and selects the one with the highest utility index (i.e., $\arg\max U(s), \forall s \in F(M)$). However, as the number of possible execution strategies grows exponentially with the number of equivalent microservices, estimating the QoS for each of them may take too long.

The approximation heuristic first sorts the equivalent microservices by their utility values (i.e., the microservices appear in the order of their overall performance). The initial execution strategy only includes the first microservice from that list. Then, in each iteration, the first microservice on the list is removed and included into the strategy, thus passing through the entire list.

Both algorithms generate strategies that contain all M equivalent microservices. Another generation heuristics could generate strategies that contain only a subset of these microservices. The exhaustive search can include all possible strategies $F'(M)$ (with 1 to M microservices) instead of $F(M)$, while the approximation heuristic can terminate when including a microservice into a strategy fails to improve the utility index. However, as the execution resources in an environment may change over time, executing a generated plan that includes only a subset of equivalent microservices may cause the remaining microservices to stay excluded from being executed. If an originally included microservice becomes unavailable, the strategy generator may fail to switch to an alternative superior strategy, due to the lack of historical execution data for the microservices excluded from the original strategy.

6.4 Reference Implementation and Evaluation

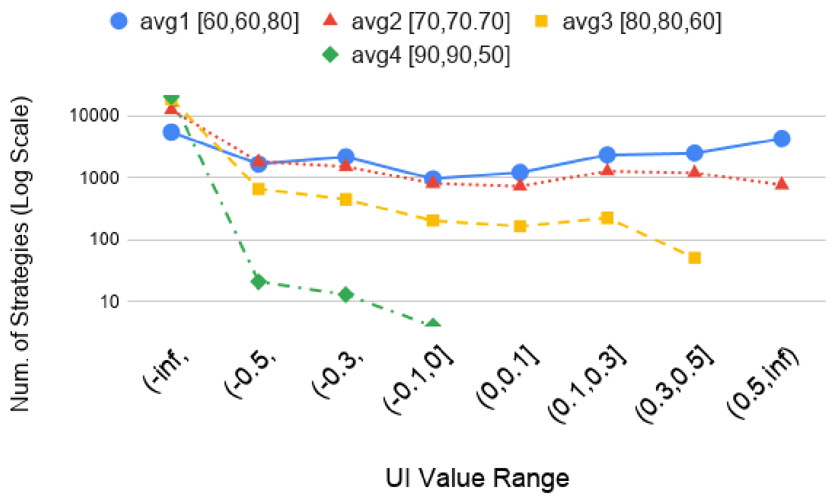


Figure 6.4: Exp1: Varying avg [c, l, r]

Our evaluation seeks answers to the following questions:

- Does changing execution strategies substantially impact QoS?
- Is our QoS estimation accurate? How does our generated strategy compare with the predefined strategies in terms of their estimated QoS?
- How does the approximation heuristic perform compared with the exhaustive search?
- How does our system perform in real setups? Does it outperform MOLE in dissimilar edge environments?

In the following, our evaluation confirms that our QoS estimation can reliably predict the expected service performance. Compared with the predefined strategies, our generated strate-

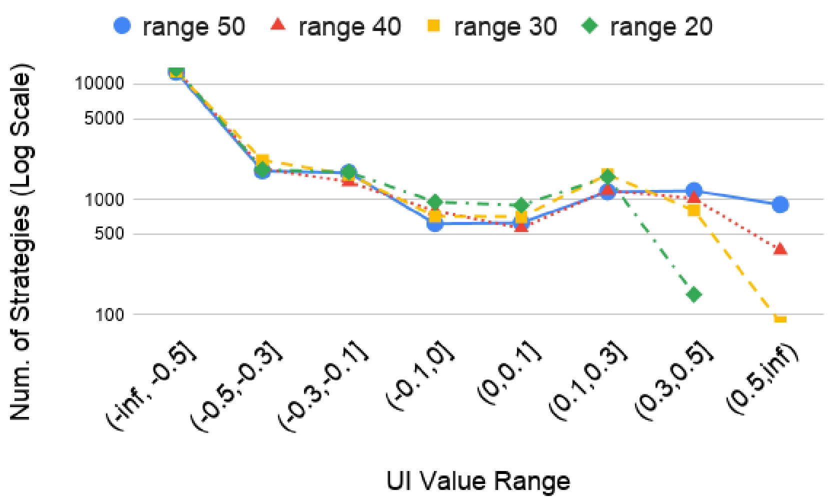


Figure 6.5: Exp2: Varying QoS Range(Δ)

gies increase the ratio of QoS-satisfied services by 2× for fewer than 5 equivalent microservices, and by 2.6 × for 5 to 10 equivalent microservices. In a given edge environment, our system outperforms MOLE in terms of cost, latency, and reliability by 31%, 52%, and 4%, respectively. Besides, our system dynamically optimizes the overall QoS by adapting to the resource changes of edge environments.

6.4.1 Simulation

The simulation runs on a ThinkCentre M900 Tiny desktop (i7-6700T CPU and 32G memory). We randomly assign QoS values to a number of equivalent microservices.

Utility of all Possible Execution Strategies

As shown in Table 6.3, we conduct three sets of experiments, $exp1$, $exp2$, $exp3$, each with a number of configurations. We use [c, l, r] to denote the average value of cost, latency, and reliability, respectively, and use Δ to denote the value range (e.g., $cost = rand(c - \frac{\Delta}{2}, c +$

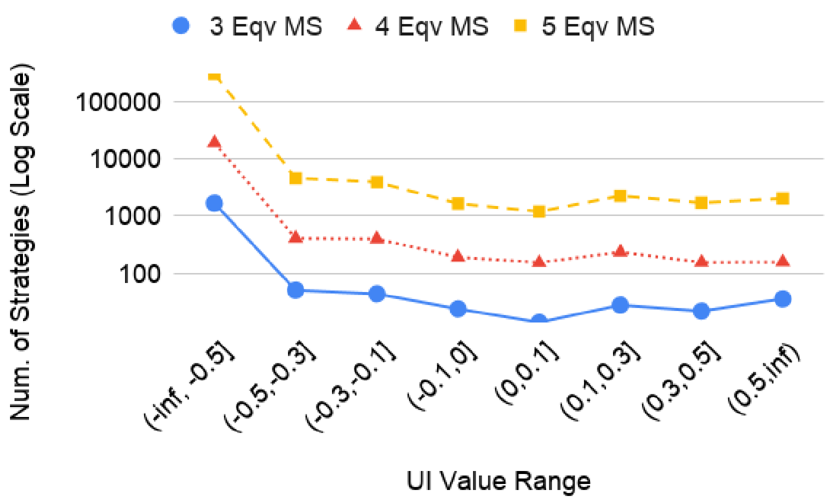


Figure 6.6: Exp3: Varying Number of Eqv. Microservices

$\frac{\Delta}{2}$)). For each configuration, we simulate 100 services. The QoS requirements in all three experiments are $Q_c = 100$ (units), $Q_l = 100$ (ms), $Q_r = 97$ (%).

For *exp*1, *exp*2, and *exp*3, Fig. 6.4, 6.5, and 6.6 show the utility distribution of all possible strategies for all randomly generated 100 services in each configuration, respectively. Different lines in each graph denote different configurations. In general, we observe that for all configurations, different execution strategies lead to vastly dissimilar utilities. With higher average QoS, higher Δ (the varying range of QoS), and more equivalent microservices, more execution strategies show higher utility index values.

Correctness of QoS Estimation

We randomly select 100 execution strategies from different configurations, and compare their execution performance with our QoS estimations. We use `system.sleep` to imitate each microservice's execution latency, with each strategy executed 300 times. To filter out the costs of scheduling multi-threaded executions, we use "second" as the latency unit of

Exp ID	Config ID	Num of Eqv MS	avg c, l, r	Δ
exp1	1	4	60, 60, 80	50
	2		70, 70, 70	
	3		80, 80, 60	
	4		90, 90, 50	
exp2	1	4	70, 70, 70	50
	2			40
	3			30
	4			20
exp3	1	3	90, 90, 50	100
	2	4		
	2	5		

Table 6.3: Simulation Configurations

microservices. For example, to verify the execution latency of $a * b * c$ with the QoS settings in Section III.C, we set the average execution time of a, b, and c to 10, 90, and 70 seconds, respectively, and then observe the average overall execution latency of 69.43 seconds. For the other executions, the difference between the average execution latency and our estimations are less than 1%, thus confirming the correctness of our QoS estimation.

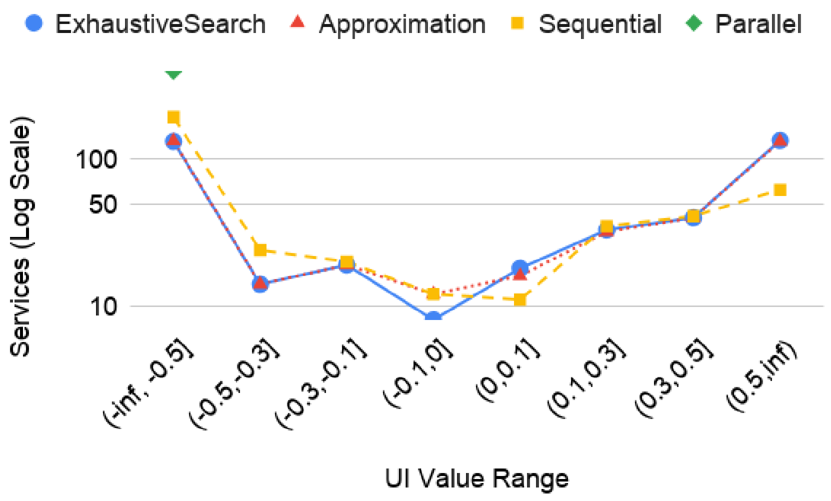


Figure 6.7: UI Distribution for Exp1

Then, we calculate the utility values of strategies generated by the exhaustive search and

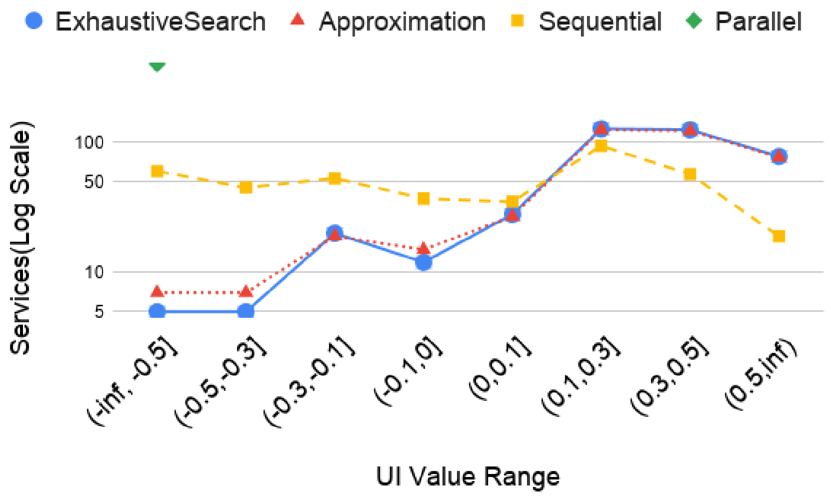


Figure 6.8: UI Distribution for Exp2

approximation heuristics, and those of the predefined sequential and parallel strategies, as shown in Fig. 6.7, 6.8, and 6.9. From the UI value distribution, we observe that: 1) our strategies obviously outperform the predefined strategies for all three experiments, as more of their utilities fall into the range of high values; 2) the exhaustive search and *Approximation* produce strategies with comparable performance in terms of their utility values. Fig. 6.10 and 6.11 show the number of services whose QoS requirements are satisfied and the average utility values of various generation heuristics under each configuration. Compared with the predefined strategies, our heuristic increases the ratio of QoS-satisfied services by an average of $2\times$.

Besides, from Fig.6.10 we also observe that the overall performance of the generated strategies is impacted by the number of equivalent microservices and their average performance, but is not impacted by the QoS range (Δ) of these microservices.

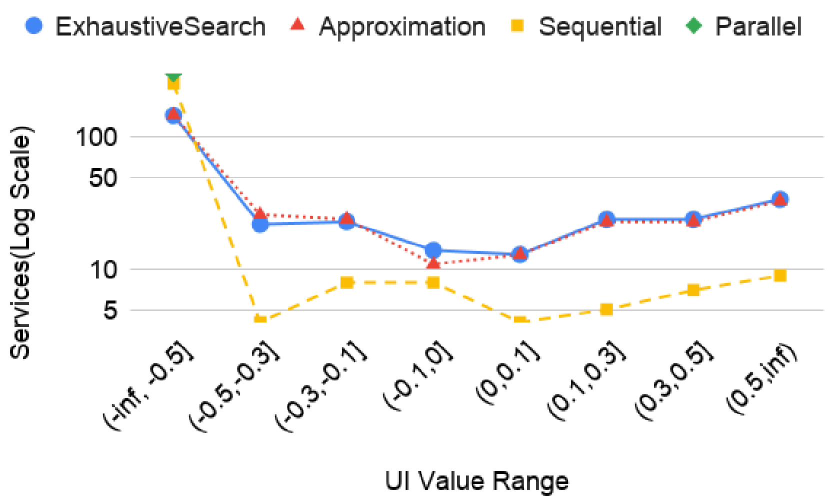


Figure 6.9: UI Distribution for Exp3

Comparing Exhaustive Search and Approximation

To evaluate how our generation heuristic scales, Fig. 6.12, 6.13, and 6.14 show the performance of edge services with more than 5 equivalent microservices. Fig. 6.12 shows the generation time of different algorithms. With the increase of equivalent microservices in an edge service, the exhaustive search's time increases exponentially, while the time taken by the approximation heuristic and that by the default strategy (either sequential or parallel, represented as a tree) increase only moderately. Fig. 6.13 and 6.14 show the number of QoS-satisfied services and the average utility values of different strategies. Hence, as the number of equivalent microservices increases, our generator continues outperforming the predefined strategies (2.6 × QoS-satisfied services) without incurring much additional execution latency (10% extra time).

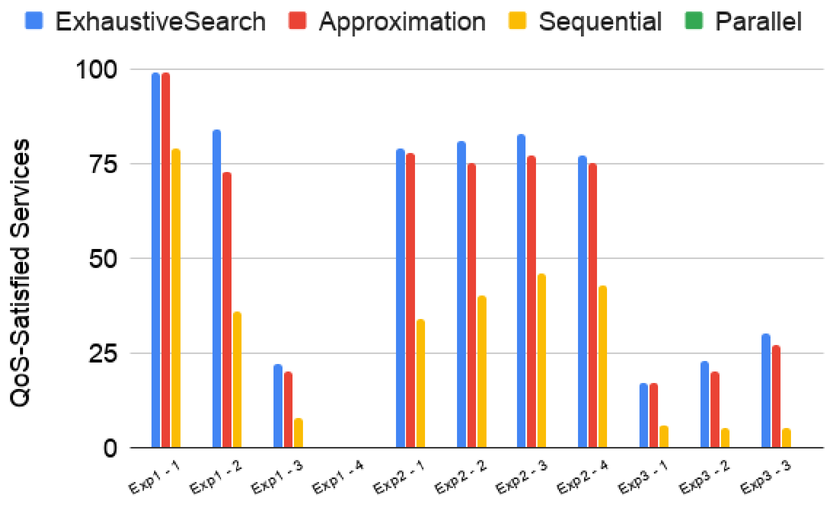


Figure 6.10: Number of Services with Fully Satisfied QoS of Different Generated Strategies

6.4.2 System Performance

To support the cross-platform deployment on edge gateways, our runtime system is implemented in Java. In our experimental setup, a ThinkCentre M900 Tiny desktop (i7-6700T CPU and 32G memory) serves as the gateway, while a Raspberry Pi 3 (BCM2837 CPU and 1G RAM) and two ThinkCentre M92p Tiny desktops (i5-6500T CPU and 8G memory) serve as edge devices. Each edge device registers its available microservices and their usage costs with the gateway.

To compare with MOLE, we reimplement its evaluation use cases. Three microservices are deployed to detect the ambient temperature, including 1) read a DS1820 temperature sensor; (`readTempSensor`) 2) read a CPU temperature sensor and estimate the environmental temperature [84] (estTemp); 3) query a web service for the location of the current IP address, and query another web service for the location's temperature (readLocTemp). We deploy `readTempSensor` on the Raspberry Pi with a DS1820 sensor connected via GPIO. The execution time for reading the DS1820 sensor is around 950ms, so the microservice reads the

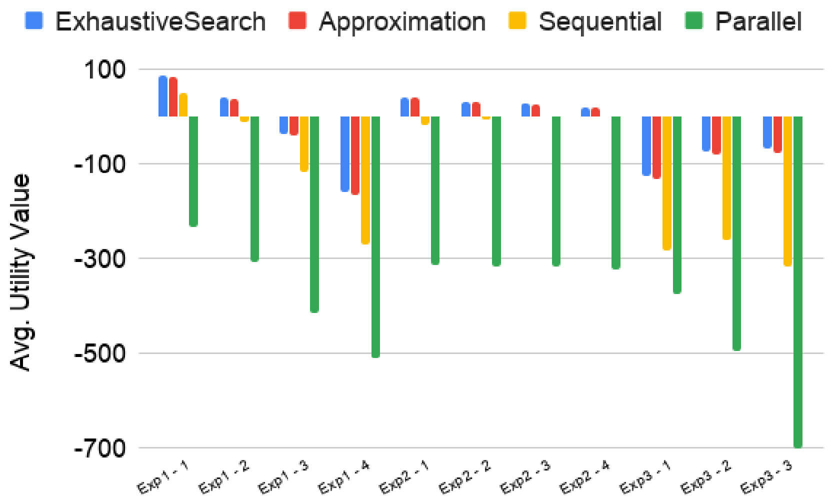


Figure 6.11: Average Utility Values of Generated Strategies

sensor every 30 seconds, caches the results, and uses the cached readings as output. `estTemp` and `readLocTemp` are deployed separately on the two M92p Tiny desktops.

We simulate 100 service invocations per a time slot. In the first time slot, the gateway has no previous microservice execution history, so it follows the **default** speculative parallel strategy. In the next time slots, the gateway uses the execution records in the previous time slot to generate execution strategies and execute them. We set the reliability of these three microservices to 70%, and their cost to 50. The generated strategy is "`readTempSensor-estTemp-readLocTemp`". Table 6.4 shows the execution results. We observe that: 1) the measured QoS of the generated strategy is better than that of the default strategy; 2) the difference between the measured QoS and the estimated QoS is minor.

QoS	Default Strategy	Estimation of Gen. Strategy	Measured
cost	100	70	69
latency	163	81	78
reliability	94	97	98

Table 6.4: Execution Results of Setting 1

We further show how our system adapts to the changes in microservice QoS in an edge

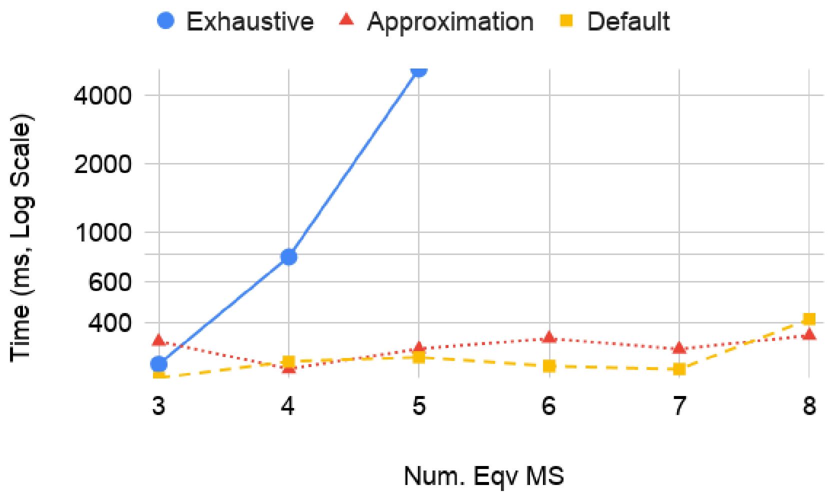


Figure 6.12: Strategy Generation Time

environment. We adopt the microservice QoS and service QoS requirements of the setting above, and emulate the resource change by: 1) after being executed 230 times (a randomly selected number), the reliability of `readTempSensor` drops to 20%; 2) after being executed 430 times, the reliability of `readTempSensor` recovers back to 70%. Figure 6.15 shows the QoS of different time slots, each comprising 100 executions. The execution strategy generated after executing the default speculative parallel strategy is `readTempSensor-estTemp-readLocTemp` . At time slot 1, the reliability of `readTempSensor` drops to 20%. Hence, the execution strategy for slots 2 to 5 is `estTemp-readTempSensor-readLocTemp`. Then, the reliability of `readTempSensor` recovers at slot 5, so the execution strategy for slots 6 and 7 gets back to the previous strategy. We observe that: 1) the QoS of slots 2, 3 and 4 is better than that of slot 1; 2) the QoS of slots 6 and 7 is better than that of slot 5. This experiment shows that switching between the execution strategies of equivalent microservices of an edge service indeed adapts to the QoS fluctuations of these microservices.

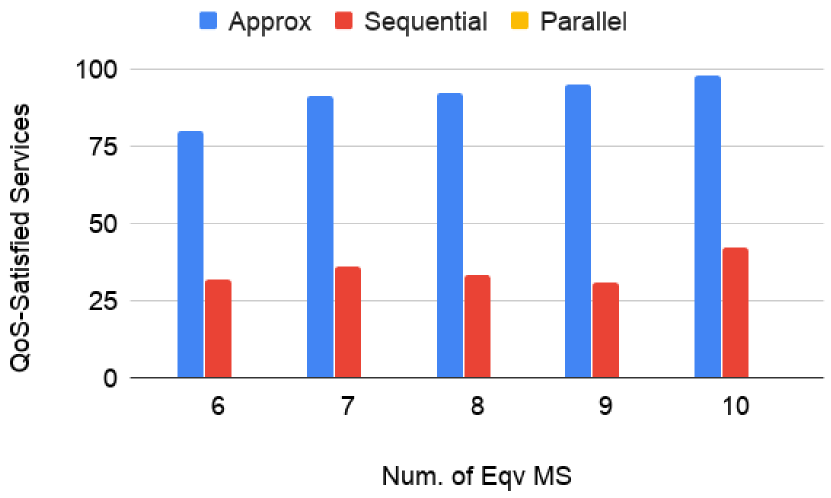


Figure 6.13: QoS Satisfaction Ratio of Strategies for More than 5 Eqv MS

6.5 Related Work

The resources in edge environments are typically scarce, unreliable, and dynamic. To guarantee QoS with scarce resources, most recent edge system designs [111, 141, 174, 193, 194] take advantage of remote resources by offloading computationally intensive tasks to the cloud or nearby edges. To improve reliability and handle unpredictable failures in edge networks, [63] deploys redundant resources as fail-over backups. To adapt to resource dynamicity across edge environments, [60, 72, 175] dynamically adjusts the computational load of edge-based executions by controlling their runtime parameters to fit the available resource budgets, while [93, 190] provide uniform interface to abstract dissimilar hardware and their capabilities.

However, none of the aforementioned designs would be applicable under the following constraints. Remote resources cannot be relied on in the absence of network connectivity or when the local context is required; redundant identical resources may not always be deployed in resource scarce environments; configuring executions self-adaptively may incur runtime failures. To the best our knowledge, our own MOLE[153] is the first attempt to exploit

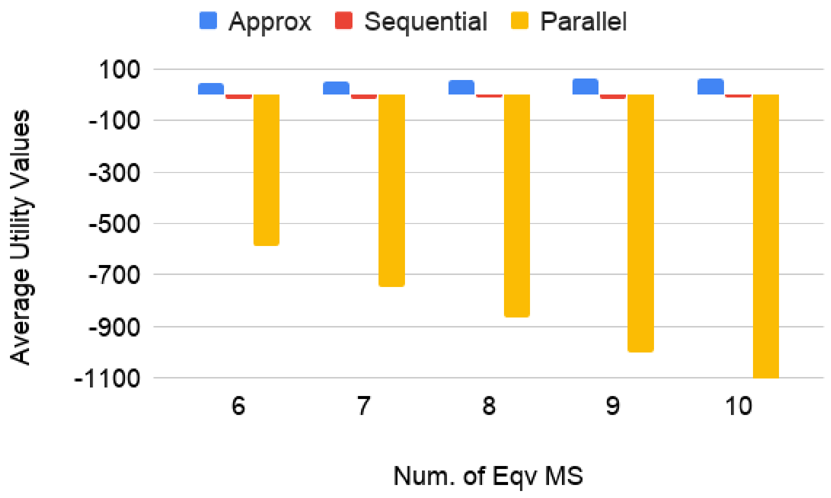


Figure 6.14: Utility Values of Strategies for More than 5 Eqv MS

the widely occurring resource/functionality equivalence in edge environments to address the resource scarcity and execution unreliability issues. Instead of relying on identical resources to recover from failures, MOLE relies on resources that provide equivalent functionalities. However, MOLE cannot customize execution strategies on demand to adapt for dissimilar resources across edge environments.

Having not been explored in edge computing, web service compositions apply the combined execution of equivalent services [20, 51, 57], albeit with crude-grained QoS estimation methods. Our work improves the precision of estimating the QoS of execution strategies. To compose equivalent web services, a utility function in [51] normalizes the utility of a QoS attribute by considering its lowest and highest values across all services. In contrast, our utility index normalizes the utility of a QoS attribute in accordance with its QoS requirements, so as to avoid being impacted by the QoS attribute outliers of equivalent microservices.

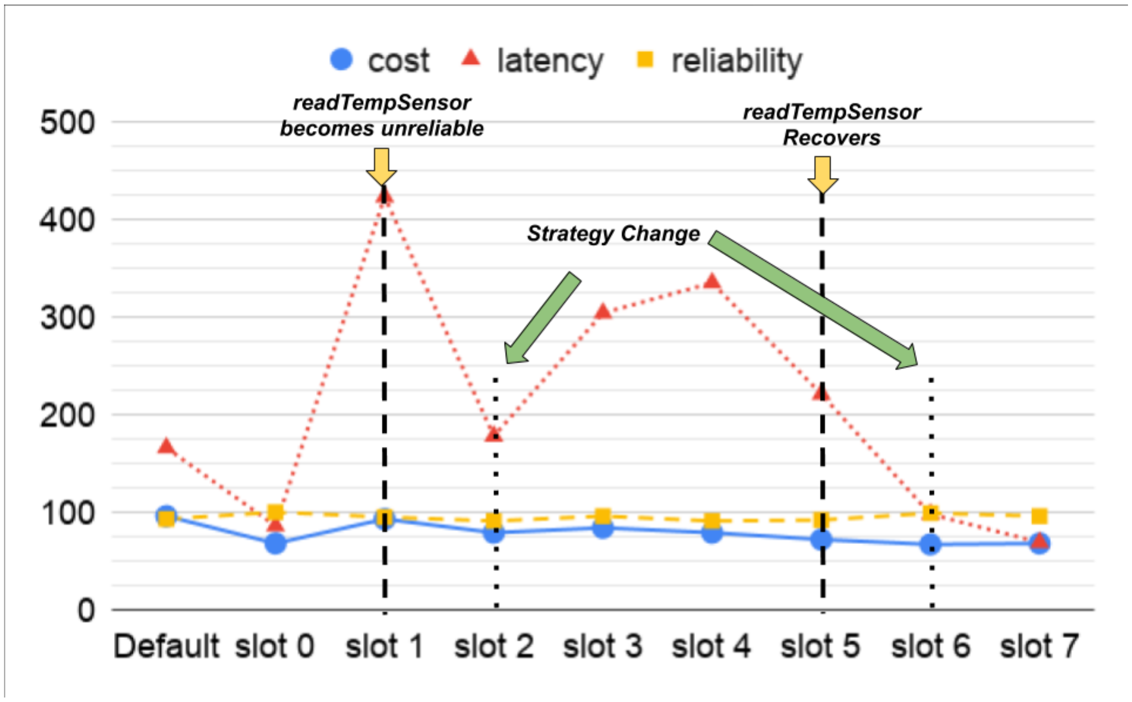


Figure 6.15: Average QoS in Different Runs

6.6 Conclusion

This chapter introduces a novel system design that provides edge services with best effort QoS. Our design improves reliability by executing equivalent functionalities and adapts to resource dissimilarity by varying execution strategies. Through a feedback loop, our design generates environment-specific strategies on demand. As an alternative to adding additional resources, our system design provides best effort edge services by better utilizing the unreliable and dynamic resources at hand. For future work, we plan to apply our system design to improve the scalability and trustworthiness of edge services. Edge systems could invoke equivalent microservices to process multiple concurrent service requests that rely on the same execution resources but are bound by their scarcity, or to protect from malicious devices that return fake results.

Chapter 7

Summary and Future Work

As compared to cloud computing, edge computing provides computational, sensor, and storage capabilities by utilizing resources at the edge of the network, thereby reducing the network traffic and providing context-awareness. However, unlike cloud computing, which relies on resources hosted by server farms and can be allocated to satisfy the demand at runtime, edge computing relies on the resources in the vicinity provided by heterogeneous devices, ranging from stationary edge servers to mobile, IoT, and energy harvesting devices. The following obstacles stand on the way of developing edge computing applications: 1) the heterogeneity of resource provisioning devices, including different device capability, various operating systems and execution platforms, dissimilar communication interfaces, makes it hard for developers to leverage their resources; 2) the high failure ratio of edge executions, caused by device mobility and low energy status, or other environmental related factors, makes edge computing unreliable; 3) the scarcity of resources makes it hard for edge systems to handle execution requests efficiently; 4) the dynamicity of resources across edge environments makes it impossible to guarantee the QoS of edge executions by relying on the existence of a standard set of resources.

To tackle the aforementioned problems, this dissertation research has two main thrusts: first, it designs and develops system architecture and programming support for providing edge services by using heterogeneous and ever changing edge devices. Secondly, it systemically studies how to leverage equivalent functionalities to enhance the reliability, efficiency, and adaptiveness of edge-based services.

With the rapid growth of IoT, wearable computing, and smart home setups and applications, there is growing demand for novel programming and system support for mobile and edge computing applications. The resource provisioning of edge systems is fundamentally different from that of cloud systems, as edge systems rely on the available resources provided by a variety of stakeholders at runtime, instead of allocating resources pre-deployed by vendors. The heterogeneity, resource scarcity, and unreliability of edge devices make the performance tuning methodologies for cloud-based distributed systems no longer applicable to edge-based systems. Meanwhile, the prevalence of equivalent functionalities in this domain provides unique but underestimated opportunities for enhancing the performance of edge applications. To enhance mobile edge computing by systemically leveraging equivalent functionalities, several potential future research directions include: (a) advancing the theoretical models for how equivalent executions can enhance the scalability and trustworthiness of edge systems: as equivalent functionalities consume dissimilar resources, edge systems can adjust the request handling methods to accommodate to the available resources in an edge system; besides, by executing multiple equivalent functionalities and comparing their output, an edge system can detect those devices that report untrustworthy results; (b) integrating equivalent executions into major cluster-computing platforms (i.e., docker swarm) to provide built-in support for enhancing the reliability, efficiency, adaptiveness, scalability, and trustworthiness of edge-based services.

Bibliography

[1] Nabil R Adam, Vijayalakshmi Atluri, and Wei-Kuang Huang. Modeling and analysis of workflows using petri nets. *Journal of Intelligent Information Systems*, 10(2):131–158, 1998.

[2] Yasser Aldwyan and Richard O Sinnott. Latency-aware failover strategies for containerized web applications in distributed clouds. *Future Generation Computer Systems*, 101:1081–1095, 2019.

[3] Mohammad Alrifai and Thomas Risse. Combining global optimization with local selection for efficient qos-aware service composition. In *Proceedings of the 18th international conference on World wide web*, pages 881–890. ACM, 2009.

[4] Nuha Alshuqayran, Nour Ali, and Roger Evans. A systematic mapping study in microservice architecture. In *Service-Oriented Computing and Applications (SOCA), 2016 IEEE 9th International Conference on*, pages 44–51. IEEE, 2016.

[5] Brian Amento, Bharath Balasubramanian, Robert J Hall, Kaustubh Joshi, Gueyoung Jung, and K Hal Purdy. Focusstack: Orchestrating edge clouds using location-based focus of attention. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 179–191. IEEE, 2016.

[6] Ardalan Amiri Sani, Kevin Boos, Min Hong Yun, and Lin Zhong. Rio: a system solution for sharing i/o between mobile systems. In *MobiSys'14*, pages 259–272. ACM, 2014.

[7] Ardalan Amiri Sani, Kevin Boos, Min Hong Yun, and Lin Zhong. Rio: a system

solution for sharing i/o between mobile systems. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services (MobiSys'14)*, pages 259–272. ACM, 2014.

[8] Hany Atlam, Robert Walters, and Gary Wills. Fog computing and the internet of things: a review. *big data and cognitive computing*, 2(2):10, 2018.

[9] Saurabh Bagchi, Muhammad-Bilal Siddiqui, Paul Wood, and Heng Zhang. Dependability in edge computing. *Communications of the ACM*, 63(1):58–66, 2019.

[10] Venkatraman Balasubramanian, Moayad Aloqaily, Faisal Zaman, and Yaser Jararweh. Exploring computing at the edge: a multi-interface system architecture enabled mobile device cloud. In *2018 IEEE 7th International Conference on Cloud Networking (CloudNet)*, pages 1–4. IEEE, 2018.

[11] Fehmi Ben Abdesslem and Anders Lindgren. Demo: mobile opportunistic system for experience sharing (moses) in indoor exhibitions. In *Proceedings of the 20th annual international conference on Mobile computing and networking (MobiCom'14)*, pages 267–270. ACM, 2014.

[12] Christian Berkhoff, Sergio F Ochoa, José A Pino, Jesus Favela, Jonice Oliveira, and Luis A Guerrero. Clairvoyance: A framework to integrate shared displays and mobile computing devices. *Future Generation Computer Systems*, 34:190–200, 2014.

[13] Ketan Bhardwaj, Sreenidhy Sreepathy, Ada Gavrilovska, and Karsten Schwan. Ecc: Edge cloud composites. In *MobileCloud'14*, pages 38–47. IEEE, 2014.

[14] Ketan Bhardwaj, Ming-Wei Shih, Pragya Agarwal, Ada Gavrilovska, Taesoo Kim, and Karsten Schwan. Fast, scalable and secure onloading of edge functions using airbox. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 14–27. IEEE, 2016.

[15] Aabhas Bhatia, Shuangyi Li, Zheng Song, and Eli Tilevich. Exploiting equivalence to efficiently enhance the accuracy of cognitive services. In *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 143–150. IEEE, 2019.

[16] Mario Bisignano, Giuseppe Di Modica, and Orazio Tomarchio. Jmobipeer: a middleware for mobile peer-to-peer computing in manets. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems Workshops (ICDCS'05 Workshop)*, pages 785–791. IEEE, 2005.

[17] Antonio Brogi and Stefano Forti. QoS-aware deployment of IoT applications through the fog. *IEEE Internet of Things Journal*, 4(5):1185–1192, 2017.

[18] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. Engineering self-adaptive systems through feedback loops. In *Software engineering for self-adaptive systems*, pages 48–70. Springer, 2009.

[19] Mauro Caporuscio, P-G Raverdy, and Valerie Issarny. ubisoap: A service-oriented middleware for ubiquitous networking. *Services Computing, IEEE Transactions on*, 5 (1):86–98, 2012.

[20] Valeria Cardellini, Emiliano Casalicchio, Vincenzo Grassi, Stefano Iannucci, Francesco Lo Presti, and Raffaela Mirandola. Moses: A framework for qos driven runtime adaptation of service-oriented systems. *IEEE Transactions on Software Engineering*, 38(5):1138–1159, 2011.

[21] Francisco Carpio, Admela Jukan, Roman Sosa, and Ana Juan Ferrer. Engineering a QoS provider mechanism for edge computing with deep reinforcement learning. *arXiv preprint arXiv:1905.00785*, 2019.

[22] Alessandro Carrega, Matteo Repetto, Giorgio Robino, and Giancarlo Portomauro. Openstack extensions for QoS and energy efficiency in edge computing. In *2018 Third International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 50–57. IEEE, 2018.

[23] Antonio Carzaniga, Alessandra Gorla, Nicolo Perino, and Mauro Pezze. Automatic workarounds: Exploiting the intrinsic redundancy of web applications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(3):16, 2015.

[24] Fabio Casati, Stefano Ceri, Barbara Pernici, and Giuseppe Pozzi. Conceptual modeling of workflows. In *International Conference on Conceptual Modeling*, pages 341–354. Springer, 1995.

[25] Alberto Huertas Celdrán, Félix J García Clemente, James Weimer, and Insup Lee. Ice++: improving security, qos, and high availability of medical cyber-physical systems through mobile edge computing. In *2018 IEEE 20th International Conference on e-Health Networking, Applications and Services (Healthcom)*, pages 1–8. IEEE, 2018.

[26] Kongyang Chen, Chen Wang, Zhimeng Yin, Hongbo Jiang, and Guang Tan. Slide: Towards fast and accurate mobile fingerprinting for wi-fi indoor positioning systems. *IEEE Sensors Journal*, 18(3):1213–1223, 2017.

[27] Xu Chen. Decentralized computation offloading game for mobile cloud computing. *IEEE TPDS*, 26(4):974–983, 2015.

[28] Xu Chen, Lei Jiao, Wenzhong Li, and Xiaoming Fu. Efficient multi-user computation offloading for mobile-edge cloud computing. *IEEE/ACM TON*, 2015.

[29] Bin Cheng, Gürkan Solmaz, Flavio Cirillo, Ernö Kovacs, Kazuyuki Terasawa, and

Atsushi Kitazawa. Fogflow: Easy programming of iot services over cloud and edges for smart cities. *IEEE IoT Journal*, 5(2):696–707, 2018.

[30] Jimin Cheon, Jeonghwan Lee, Inhee Lee, Youngcheol Chae, Youngsin Yoo, and Gunhee Han. A single-chip cmos smoke and temperature sensor for an intelligent fire detector. *IEEE Sensors Journal*, 9(8):914–921, 2009.

[31] Mung Chiang and Tao Zhang. Fog and iot: An overview of research opportunities. *IEEE Internet of Things Journal*, 3(6):854–864, 2016.

[32] Xuewen Cui and Wu-chun Feng. Iterative machine learning (iterml) for effective parameter pruning and tuning in accelerators. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*, pages 16–23. ACM, 2019.

[33] Eduardo da Silva and Luiz Carlos P Albini. Middleware proposals for mobile ad hoc networks. *Journal of Network and Computer Applications*, 43:103–120, 2014.

[34] Mieso K Denko, Elhadi Shakshuki, and Haroon Malik. A mobility-aware and cross-layer based middleware for mobile ad hoc networks. In *Proceedings of the 21st International Conference on Advanced Information Networking and Applications (AINA'07)*, pages 474–481. IEEE, 2007.

[35] Western Digital. How edge computing will revolutionize mobile gaming, 2019. https://datamakespossible.westerndigital.com/edge-computing-mobile-gaming/.

[36] Adam Dou, Vana Kalogeraki, Dimitrios Gunopulos, Taneli Mielikainen, and Ville H Tuulos. Misco: a mapreduce framework for mobile systems. In *Proceedings of the 3rd International Conference on Pervasive Technologies related to Assistive Environments*, page 32. ACM, 2010.

[37] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017.

[38] Utsav Drolia, Katherine Guo, and Priya Narasimhan. Precog: Prefetching for image recognition applications at the edge. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, pages 1–13, 2017.

[39] Daniel J Dubois, Yosuke Bando, Konosuke Watanabe, and Henry Holtzman. Shair: Extensible middleware for mobile peer-to-peer resource sharing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (FSE'13)*, pages 687–690. ACM, 2013.

[40] John Edstrom and Eli Tilevich. Improving the survivability of restful web applications via declarative fault tolerance. *Concurrency and Computation: Practice and Experience*, pages n/a–n/a, 2014. ISSN 1532-0634. doi: 10.1002/cpe.3197. URL http://dx.doi.org/10.1002/cpe.3197.

[41] Hesham El-Sayed, Sharmi Sankar, Mukesh Prasad, Deepak Puthal, Akshansh Gupta, Manoranjan Mohanty, and Chin-Teng Lin. Edge of things: The big picture on the integration of edge, iot and the cloud in a distributed computing environment. *IEEE Access*, 6:1706–1717, 2017.

[42] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[43] flame sensor. Special issue "sensors for fire detection". https://www.mdpi.com/journal/sensors/special_issues/SFD, 2016. [Online; accessed 13-Feb-2019].

[44] Chien-Liang Fok, Gruia-Catalin Roman, and Gregory Hackmann. A lightweight coordination middleware for mobile computing. In *Coordination Models and Languages*, pages 135–151. Springer, 2004.

[45] Abhrajit Ghosh, Shih-wei Li, C Jason Chiang, Ritu Chadha, Kimberly Moeltner, Syeed Ali, Yogeeta Kumar, and Rocio Bauer. Qos-aware adaptive middleware (qam) for tactical manet applications. In *MILCOM'10*, pages 178–183. IEEE, 2010.

[46] Nam Ky Giang, Michael Blackstock, Rodger Lea, and Victor CM Leung. Developing iot applications in the fog: a distributed dataflow approach. In *Internet of Things (IOT), 2015 5th International Conference on the*, pages 155–162. IEEE, 2015.

[47] Nam Ky Giang, Rodger Lea, Michael Blackstock, and Victor CM Leung. Fog at the edge: Experiences building an edge computing platform. In *IEEE EDGE'18*, pages 9–16. IEEE, 2018.

[48] Li Gong. Jxta: A network programming environment. *Internet Computing, IEEE*, 5 (3):88–95, 2001.

[49] Maria Gorlatova, John Sarik, Guy Grebla, Mina Cong, Ioannis Kymissis, and Gil Zussman. Movers and shakers: Kinetic energy harvesting for the internet of things. *IEEE Journal on Selected Areas in Communications*, 33(8):1624–1639, 2015.

[50] Huipeng Guo, Jinpeng Huai, Huan Li, Ting Deng, Yang Li, and Zongxia Du. Angel: Optimal configuration for high available service composition. In *IEEE International Conference on Web Services (ICWS 2007)*, pages 280–287. IEEE, 2007.

[51] Yan Guo, Shangguang Wang, Kok-Seng Wong, and Myung Ho Kim. Skyline service selection approach based on qos prediction. *International Journal of Web and Grid Services*, 13(4):425–447, 2017.

[52] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. Towards wearable cognitive assistance. Technical report, DTIC Document, 2013.

[53] Kiryong Ha, Padmanabhan Pillai, Grace Lewis, Soumya Simanta, Sarah Clinch, Nigel Davies, and Mahadev Satyanarayanan. The impact of mobile multimedia applications on data center consolidation. In *2013 IEEE international conference on cloud engineering (IC2E)*, pages 166–176. IEEE, 2013.

[54] Karim Habak, Mostafa Ammar, Khaled A Harras, and Ellen Zegura. Femto clouds: Leveraging mobile devices to provide cloud service at the edge. In *CloudCom'15*, pages 9–16. IEEE, 2015.

[55] Sara Hassan and Rami Bahsoon. Microservices and their design trade-offs: A self-adaptive roadmap. In *2016 IEEE International Conference on Services Computing (SCC)*, pages 813–818. IEEE, 2016.

[56] Klaus Herrmann. Meshmd1-a middleware for self-organization in ad hoc networks. In *ICDCS'03 Workshop*, pages 446–451. IEEE, 2003.

[57] Nobuaki Hiratsuka, Fuyuki Ishikawa, and Shinichi Honiden. Service selection with combinational use of functionally-equivalent services. In *Web Services (ICWS), 2011 IEEE International Conference on*, pages 97–104. IEEE, 2011.

[58] Martin Hirzel and Guillaume Baudart. Stream processing languages and abstractions. *Encyclopedia of Big Data Technologies*, 2018.

[59] Kirak Hong, David Lillethun, Umakishore Ramachandran, Beate Ottenwälder, and Boris Koldehofe. Mobile fog: A programming model for large-scale applications on the

internet of things. In *Proceedings of the second ACM SIGCOMM workshop on Mobile cloud computing*, pages 15–20. ACM, 2013.

[60] Miao Hu, Lei Zhuang, Di Wu, Yipeng Zhou, Xu Chen, and Liang Xiao. Learning driven computation offloading for asymmetrically informed edge computing. *IEEE Transactions on Parallel and Distributed Systems*, 2019.

[61] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. Mobile edge computing—a key technology towards 5g. *ETSI White Paper*, 11(11):1–16, 2015.

[62] Yung-Li Hu, Yuo-Yu Cho, Wei-Bing Su, David SL Wei, Yennun Huang, Jiann-Liang Chen, Yi Chen, and Sy-Yen Kuo. A programming framework for implementing fault-tolerant mechanism in iot applications. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 771–784. Springer, 2015.

[63] Huawei Huang and Song Guo. Proactive failure recovery for nfv in distributed edge computing. *IEEE Communications Magazine*, 57(5):131–137, 2019.

[64] Pan Hui, Augustin Chaintreau, James Scott, Richard Gass, Jon Crowcroft, and Christophe Diot. Pocket switched networks and human mobility in conference environments. In *Proceedings of the 2005 ACM SIGCOMM workshop on Delay-tolerant networking (WDTN'05)*, pages 244–251. ACM, 2005.

[65] Chien-Chun Hung, Ganesh Ananthanarayanan, Peter Bodik, Leana Golubchik, Minlan Yu, Paramvir Bahl, and Matthai Philipose. Videoedge: Processing camera streams using hierarchical clusters. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 115–131. IEEE, 2018.

[66] Shadi Ibrahim, Hai Jin, Bin Cheng, Haijun Cao, Song Wu, and Li Qi. Cloudlet: towards mapreduce implementation on virtual machines. In *Proceedings of the 18th*

ACM international symposium on High performance distributed computing, pages 65–66, 2009.

[67] Minsung Jang, Karsten Schwan, Ketan Bhardwaj, Ada Gavrilovska, and Adhyas Avasthi. Personal clouds: Sharing and integrating networked resources to enhance end user experiences. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*, pages 2220–2228. IEEE, 2014.

[68] Ward Jaradat, Alan Dearle, and Adam Barker. A dataflow language for decentralised orchestration of web service workflows. In *Services (SERVICES), IEEE Ninth World Congress on*, pages 13–20. IEEE, 2013.

[69] Peng Jiang, John Bigham, Eliane Bodanese, and Emmanuel Claudel. Publish/subscribe delay-tolerant message-oriented middleware for resilient communication. *Communications Magazine, IEEE*, 49(9):124–130, 2011.

[70] Guo Jie, Cheng Bo, Zhao Shuai, and Chen Junliang. Cross-platform android/ios-based smart switch control middleware in a digital home. *Mobile Information Systems*, 2015, 2015.

[71] Wesley M Johnston, JR Hanna, and Richard J Millar. Advances in dataflow programming languages. *ACM comp. surveys (CSUR)*, 36(1):1–34, 2004.

[72] Albert Jonathan, Abhishek Chandra, and Jon Weissman. Locality-aware load sharing in mobile cloud computing. In *Proceedings of the10th International Conference on Utility and Cloud Computing*, pages 141–150. ACM, 2017.

[73] Brennan Jones, Kody Dillman, Richard Tang, Anthony Tang, Ehud Sharlin, Lora Oehlberg, Carman Neustaedter, and Scott Bateman. Elevating communication, col-

laboration, and shared experiences in mobile video through drones. In *DIS'16*, pages 1123–1135. ACM, 2016.

[74] Soon-Gyo Jung, Jisun An, Haewoon Kwak, Joni Salminen, and Bernard Jim Jansen. Assessing the accuracy of four popular face recognition tools for inferring gender, age, and race. In *Twelfth International AAAI Conference on Web and Social Media*, 2018.

[75] Tony Kauffmann. Solar power for raspberry pi. `https://blog.voltaicsystems.com/powering-a-raspberry-pi-from-solar-power/`, 2017.

[76] John E Kelly III and Steve Hamm. *Smart machines: IBM's Watson and the era of cognitive computing*. Columbia University Press, 2013.

[77] David Kitchin, Adrian Quark, William Cook, and Jayadev Misra. The orc programming language. In *Formal techniques for Distributed Systems*, pages 1–25. Springer, 2009.

[78] David Koll, Jun Li, and Xiaoming Fu. Soup: an online social network by the people, for the people. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 143–144. ACM, 2014.

[79] Jakub Konečnỳ, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*, 2016.

[80] Gerd Kortuem. Proem: a middleware platform for mobile peer-to-peer computing. *ACM SIGMOBILE Mobile Computing and Communications Review*, 6(4):62–64, 2002.

[81] Gerd Kortuem, Jay Schneider, Dustin Preuitt, Thaddeus G Cowan Thompson, Stephen Fickas, and Zary Segall. When peer-to-peer comes face-to-face: Collaborative peer-

to-peer computing in mobile ad-hoc networks. In *Proceedings of 1st International Conference on Peer-to-Peer Computing (P2P'01)*, pages 75–91. IEEE, 2001.

[82] Sokol Kosta, Andrius Aucinas, Pan Hui, Richard Mortier, and Xinwen Zhang. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *Infocom, 2012 Proceedings IEEE*, pages 945–953. IEEE, 2012.

[83] Niko Kotilainen, Matthieu Weber, Mikko Vapa, and Juori Vuori. Mobile chedar-a peer-to-peer middleware for mobile devices. In *Proceedings of the 3rd IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom'05 WorkShop)*, pages 86–90. IEEE, 2005.

[84] Chandra Krintz, Rich Wolski, Nevena Golubovic, and Fatih Bakir. Estimating outdoor temperature from cpu temperature for iot applications in agriculture. In *Proceedings of the 8th International Conference on the Internet of Things*, page 11. ACM, 2018.

[85] Young-Woo Kwon and Eli Tilevich. Energy-efficient and fault-tolerant distributed mobile execution. In *Proceedings of the* 32^{th} *IEEE International Conference on Distributed Computing Systems (ICDCS'12)*, pages 586–595. IEEE, 2012.

[86] Young-Woo Kwon and Eli Tilevich. Cloud refactoring: automated transitioning to cloud-based services. *Automated Software Engineering*, 21(3):345–372, 2014.

[87] Minh Le, Zheng Song, Young-Woo Kwon, and Eli Tilevich. Reliable and efficient mobile edge computing in highly dynamic and volatile environments. In *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 113–120. IEEE, 2017.

[88] Thinh Le Vinh, Samia Bouzefrane, Jean-Marc Farinone, Amir Attar, and Brian P

Kennedy. Middleware to integrate mobile devices, sensors and cloud computing. *Procedia Computer Science*, 52:234–243, 2015.

[89] Jaehun Lee, Hochul Lee, Young Choon Lee, Hyuck Han, and Sooyong Kang. Platform support for mobile edge computing. In *IEEE CLOUD'17*, pages 624–631. IEEE, 2017.

[90] Avraham Leff and James T Rayfield. Wso: Developer-oriented transactional orchestration of web-services. In *Web Services (ICWS), 2017 IEEE International Conference on*, pages 714–720. IEEE, 2017.

[91] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. Cloudcmp: comparing public cloud providers. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 1–14, 2010.

[92] Min Li, Zhaoxiang Zhang, Kaiqi Huang, and Tieniu Tan. Estimating the number of people in crowded scenes by mid based foreground segmentation and head-shoulder detection. In *Pattern Recognition, 2008. ICPR 2008. 19th International Conference on*, pages 1–4. IEEE, 2008.

[93] Yong Li and Wei Gao. Interconnecting heterogeneous devices in the personal mobile cloud. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[94] Yong Li and Wei Gao. Muvr: Supporting multi-user mobile virtual reality with resource constrained edge cloud. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 1–16. IEEE, 2018.

[95] Yujin Li and Wenye Wang. Can mobile cloudlets support mobile applications? In *INFOCOM'14*, pages 1060–1068. IEEE, 2014.

[96] Huiguang Liang, Hyong S Kim, Hwee-Pink Tan, and Wai-Leong Yeow. Where am i? characterizing and improving the localization performance of off-the-shelf mobile devices through cooperation. In *NOMS'16*, pages 375–382. IEEE, 2016.

[97] Jianqi Liu, Jiafu Wan, Bi Zeng, Qinruo Wang, Houbing Song, and Meikang Qiu. A scalable and quick-response software defined vehicular network assisted by mobile edge computing. *IEEE Communications Magazine*, 55(7):94–100, 2017.

[98] Peng Liu, Dale Willis, and Suman Banerjee. Paradrop: Enabling lightweight multi-tenancy at the network's extreme edge. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 1–13. IEEE, 2016.

[99] Shaoshan Liu, Liangkai Liu, Jie Tang, Bo Yu, Yifan Wang, and Weisong Shi. Edge computing for autonomous driving: Opportunities and challenges. *Proceedings of the IEEE*, 107(8):1697–1716, 2019.

[100] Xiaoyang Liu, Zheng Song, Edith Ngai, Jian Ma, and Wendong Wang. Pm2: 5 monitoring using images from smartphones in participatory sensing. In *Computer Communications Workshops (INFOCOM WKSHPS), 2015 IEEE Conference on*, pages 630–635. IEEE, 2015.

[101] Changchun Long, Yang Cao, Tao Jiang, and Qian Zhang. Edge computing framework for cooperative video processing in multimedia iot systems. *IEEE Transactions on Multimedia*, 20(5):1126–1139, 2017.

[102] Radhika Loomba, Ruairi de Frein, and Brendan Jennings. Selecting energy efficient cluster-head trajectories for collaborative mobile sensing. In *GLOBECOM'15*, pages 1–7. IEEE, 2015.

[103] Panagiotis Louridas. Orchestrating web services with bpel. *IEEE software*, 25(2): 85–87, 2008.

[104] Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel. Intermittent computing: Challenges and opportunities. In *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[105] Knud Lasse Lueth et al. State of the iot 2018: Number of iot devices now at 7b–market accelerating. *IoT Analytics*, 2018.

[106] Chaoying Ma and Jean Bacon. Cobea: A corba-based event architecture. In *Proceedings of the 4th conference on USENIX Conference on Object-Oriented Technologies and Systems-Volume 4*, pages 9–9. USENIX Association, 1998.

[107] Pavel Mach and Zdenek Becvar. Mobile edge computing: A survey on architecture and computation offloading. *IEEE Communications Surveys & Tutorials*, 19(3):1628–1656, 2017.

[108] Ahmed Mahfouz, Tarek M Mahmoud, and Ahmed Sharaf Eldin. A survey on behavioral biometric authentication on smartphones. *Journal of information security and applications*, 37:28–37, 2017.

[109] Redowan Mahmud, Ramamohanarao Kotagiri, and Rajkumar Buyya. Fog computing: A taxonomy, survey and future directions. In *Internet of everything*, pages 103–130. Springer, 2018.

[110] Marco Mamei, Franco Zambonelli, and Letizia Leonardi. Tuples on the air: A middleware for context-aware computing in dynamic networks. In *Proceedings of 23rd*

International Conference on Distributed Computing Systems Workshops (ICDCS'03 Workshop), pages 342–347. IEEE, 2003.

[111] Yuyi Mao, Jun Zhang, and Khaled B Letaief. Dynamic computation offloading for mobile-edge computing with energy harvesting devices. *IEEE Journal on Selected Areas in Communications*, 34(12):3590–3605, 2016.

[112] Eugene E Marinelli. Hyrax: cloud computing on mobile devices using mapreduce. Technical report, DTIC Document, 2009.

[113] R Timothy Marler and Jasbir S Arora. Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization*, 26(6):369–395, 2004.

[114] Rainer Mautz and Sebastian Tilch. Survey of optical indoor positioning systems. In *Indoor Positioning and Indoor Navigation (IPIN), 2011 International Conference on*, pages 1–7. IEEE, 2011.

[115] René Meier and Vinny Cahill. Steam: Event-based middleware for wireless ad hoc networks. In *ICDCS'02 Workshop*, pages 639–644. IEEE, 2002.

[116] microservice definition. What are microservices, 2005. http://microservices.io/.

[117] Seyed Hossein Mortazavi, Mohammad Salehe, Carolina Simoes Gomes, Caleb Phillips, and Eyal de Lara. Cloudpath: A multi-tier cloud computing framework. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, pages 1–13, 2017.

[118] Abderrahmen Mtibaa, Afnan Fahim, Khaled A Harras, and Mostafa H Ammar. Towards resource sharing in mobile device clouds: Power balancing across mobile devices. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 51–56. ACM, 2013.

[119] Amy L Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. Lime: A coordination model and middleware supporting mobility of hosts and agents. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(3):279–328, 2006.

[120] Kazuhiro Nakao and Yukikazu Nakamoto. Toward remote service invocation in android. In *Proceedings of the 9th International Conference on Ubiquitous Intelligence & Computing and 9th International Conference on Autonomic & Trusted Computing (UIC'12)*, pages 612–617. IEEE, 2012.

[121] Cisco Visual Networking. Cisco global cloud index: Forecast and methodology, 2016–2021. *White paper. Cisco Public, San Jose*, 2016.

[122] Andrés Neyem, Sergio F Ochoa, José A Pino, and Rubén Darío Franco. A reusable structural design for mobile collaborative applications. *Journal of Systems and Software*, 85(3):511–524, 2012.

[123] Zhaolong Ning, Xiangjie Kong, Feng Xia, Weigang Hou, and Xiaojie Wang. Green and sustainable cloud of things: Enabling collaborative edge computing. *IEEE Communications Magazine*, 57(1):72–78, 2018.

[124] OASIS Standard. BPEL 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html#_Toc164738514, 2007. [Accessed 13-Feb-2019].

[125] Jon Oberheide, Evan Cooke, and Farnam Jahanian. Cloudav: N-version antivirus in the network cloud. In *USENIX Security Symposium*, pages 91–106, 2008.

[126] Jörg Ott, Esa Hyytia, Pasi Lassila, Tobias Vaegs, and Jussi Kangasharju. Floating content: Information sharing in urban areas. In *Proceedings of the 2011 IEEE International Conference on Pervasive Computing and Communications (PerCom'11)*, pages 136–146. IEEE, 2011.

[127] Maria Papadopouli and Henning Schulzrinne. Design and implementation of a peer-to-peer data dissemination and prefetching tool for mobile users. In *Proceedings of the first NY Metro Area Networking workshop (NYMAN'01)*, 2001.

[128] Guilhem Paroux, Ludovic Martin, Julien Nowalczyk, and Isabelle Demeure. Transhumance: A power sensitive middleware for data sharing on mobile ad hoc networks. In *Proceedings of the 7th international Workshop on Applications and Services in Wireless Networks (ASWN'07)*, 2007.

[129] Pierluigi Plebani, Cinzia Cappiello, Marco Comuzzi, Barbara Pernici, and Sandeep Yadav. Micromais: executing and orchestrating web services on constrained mobile devices. *Software: Practice and Experience*, 42(9):1075–1094, 2012.

[130] Soheil Qanbari, Samim Pezeshki, Rozita Raisi, Samira Mahdizadeh, Rabee Rahimzadeh, Negar Behinaein, Fada Mahmoudi, Shiva Ayoubzadeh, Parham Fazlali, Keyvan Roshani, et al. Iot design patterns: Computational constructs to design, build and engineer edge applications. In *IoTDI'16*, pages 277–282. IEEE, 2016.

[131] Yuansong Qiao, Robert Nolani, Saul Gill, Guiming Fang, and Brian Lee. Thingnet: A micro-service based iot macro-programming platform over edges and cloud. In *2018 21st Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, pages 1–4. IEEE, 2018.

[132] Zhan Qiu. Enhancing response time and reliability via speculative replication and redundancy. *Ph.D. thesis*, 2016.

[133] Simone Raponi, Maurantonio Caprolu, and Roberto Di Pietro. Intrusion detection at the network edge: Solutions, limitations, and future directions. In *International Conference on Edge Computing*, pages 59–75. Springer, 2019.

[134] Pei Ren, Xiuquan Qiao, Junliang Chen, and Schahram Dustdar. Mobile edge computing–a booster for the practical provisioning approach of web-based augmented reality. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 349–350. IEEE, 2018.

[135] Marcin Rogowski, Khalid Saeed, Mariusz Rybnik, Marek Tabedzki, and Marcin Adamski. User authentication for mobile devices. In *Computer Information Systems and Industrial Management*, pages 47–58. Springer, 2013.

[136] Gruia-Catalin Roman, Radu Handorean, and Rohan Sen. Tuple space coordination across space and time. In *Coordination Models and Languages*, pages 266–280. Springer, 2006.

[137] Nick Russell, Arthur HM Ter Hofstede, Wil MP Van Der Aalst, and Nataliya Mulyar. Workflow control-flow patterns: A revised view. *BPM Center Report BPM-06-22, BPMcenter. org*, pages 06–22, 2006.

[138] Larisa Safina, Manuel Mazzara, Fabrizio Montesi, and Victor Rivera. Data-driven workflows for microservices: Genericity in jolie. In *Advanced Information Networking and Applications (AINA), 2016 IEEE 30th International Conference on*, pages 430–437. IEEE, 2016.

[139] Hooman Peiro Sajjad, Ken Danniswara, Ahmad Al-Shishtawy, and Vladimir Vlassov. Spanedge: Towards unifying stream processing over central and near-the-edge data centers. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 168–178. IEEE, 2016.

[140] Ahmed Salem and Tamer Nadeem. Colphone: A smartphone is just a piece of the puzzle. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication*, pages 263–266. ACM, 2014.

[141] Amit Samanta, Zheng Chang, and Zhu Han. Latency-oblivious distributed task scheduling for mobile edge computing. In *2018 IEEE Global Communications Conference (GLOBECOM)*, pages 1–7. IEEE, 2018.

[142] Genaro Saucedo-Tejada, Sonia Mendoza, and Dominique Decouchant. F2fmi: A toolkit for facilitating face-to-face mobile interaction. *Expert Systems with Applications*, 40 (15):6173–6184, 2013.

[143] Lorenz Schauer, Martin Werner, and Philipp Marcus. Estimating crowd densities and pedestrian flows using wi-fi and bluetooth. In *MobiQuitous 2014*, pages 171–177, 2014.

[144] Johannes M Schleicher, Michael Vogler, Christian Inzinger, Waldemar Hummer, and Schahram Dustdar. Nomads-enabling distributed analytical service environments for the smart city domain. In *2015 IEEE International Conference on Web Services (ICWS)*, pages 679–685. IEEE, 2015.

[145] Fernando Seco, Antonio R Jiménez, Carlos Prieto, Javier Roa, and Katerina Koutsou. A survey of mathematical methods for indoor localization. In *Intelligent Signal Processing, 2009. WISP 2009. IEEE International Symposium on*, pages 9–14. IEEE, 2009.

[146] Chenhua Shi, Zhiyuan Ren, Kun Yang, Chen Chen, Hailin Zhang, Yao Xiao, and Xiangwang Hou. Ultra-low latency cloud-fog computing for industrial internet of things. In *2018 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1–6. IEEE, 2018.

[147] Cong Shi, Vasileios Lakafosis, Mostafa H Ammar, and Ellen W Zegura. Serendipity: enabling remote computing among intermittently connected mobile devices. In *MobiHoc'12*, pages 145–154. ACM, 2012.

[148] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[149] Olena Skarlat, Matteo Nardelli, Stefan Schulte, and Schahram Dustdar. Towards qos-aware fog service placement. In *Fog and Edge Computing (ICFEC), 2017 IEEE 1st International Conference on*, pages 89–96. IEEE, 2017.

[150] Hyungjoo Song, Daeyoung Kim, Kangwoo Lee, and Jongwoo Sung. Upnp-based sensor network management architecture. In *Proc. International Conference on Mobile Computing and Ubiquitous Networking*, 2005.

[151] Zheng Song and Eli Tilevich. Pmdc: Programmable mobile device clouds for convenient and efficient service provisioning. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 202–209. IEEE, 2018.

[152] Zheng Song and Eli Tilevich. Equivalence-enhanced microservice workflow orchestration to efficiently increase reliability. In *2019 IEEE International Conference on Web Services (ICWS)*, pages 426–433. IEEE, 2019.

[153] Zheng Song and Eli Tilevich. A programming model for reliable and efficient edge-based execution under resource variability. In *2019 IEEE International Conference on Edge Computing (EDGE)*, pages 64–71. IEEE, 2019.

[154] Zheng Song, Sanchit Chadha, Antuan Byalik, and Eli Tilevich. Programming support for sharing resources across heterogeneous mobile devices. In *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 105–116. IEEE, 2018.

[155] Hui Sun, Weisong Shi, Xu Liang, and Ying Yu. Vu: Edge computing-enabled video

usefulness detection and its application in large-scale video surveillance systems. *IEEE Internet of Things Journal*, 2019.

[156] Sanjib Sur, Teng Wei, and Xinyu Zhang. Autodirective audio capturing through a synchronized smartphone array. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 28–41. ACM, 2014.

[157] Tarik Taleb, Konstantinos Samdanis, Badr Mada, Hannu Flinck, Sunny Dutta, and Dario Sabella. On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration. *IEEE Communications Surveys & Tutorials*, 19(3):1657–1681, 2017.

[158] Tian Huat Tan, Manman Chen, Jun Sun, Yang Liu, Étienne André, Yinxing Xue, and Jin Song Dong. Optimizing selection of competing services with probabilistic hierarchical refinement. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 85–95. IEEE, 2016.

[159] EdgeX Team. Edgex foundry, 2019. https://www.edgexfoundry.org/.

[160] Eli Tilevich and Young-Woo Kwon. Cloud-based execution to improve mobile application energy efficiency. *Computer*, 47(1):75–77, 2014.

[161] Mauro Tortonesi, Marco Govoni, Alessandro Morelli, Giulio Riberto, Cesare Stefanelli, and Niranjan Suri. Taming the iot data deluge: An innovative information-centric service model for fog computing applications. *Future Generation Computer Systems*, 93:888–902, 2019.

[162] Rahmadi Trimananda, Ali Younis, Bojun Wang, Bin Xu, Brian Demsky, and Guoqing Xu. Vigilia: Securing smart home edge computing. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 74–89. IEEE, 2018.

[163] Wei-Tek Tsai, Xin Sun, and Janaka Balasooriya. Service-oriented cloud computing architecture. In *2010 seventh international conference on information technology: new generations*, pages 684–689. IEEE, 2010.

[164] Carlo Vallati, Antonio Virdis, Enzo Mingozzi, and Giovanni Stea. Mobile-edge computing come home connecting things in future smart homes using lte device-to-device communications. *IEEE Consumer Electronics Magazine*, 5(4):77–83, 2016.

[165] Wil MP van Der Aalst, Arthur HM Ter Hofstede, Bartek Kiepuszewski, and Alistair P Barros. Workflow patterns. *Distributed and parallel databases*, 14(1):5–51, 2003.

[166] Maarten Van Steen, Philip Homburg, and Andrew S Tanenbaum. Globe: A wide-area distributed system. *IEEE concurrency*, 7(1):70–78, 1999.

[167] Blesson Varghese and Rajkumar Buyya. Next generation cloud computing: New trends and research directions. *arXiv preprint arXiv:1707.07452*, 2017.

[168] Blesson Varghese, Nan Wang, Sakil Barbhuiya, Peter Kilpatrick, and Dimitrios S Nikolopoulos. Challenges and opportunities in edge computing. In *IEEE SmartCloud*, pages 20–26. IEEE, 2016.

[169] Nicolas Viennot, Mathias Lécuyer, Jonathan Bell, Roxana Geambasu, and Jason Nieh. Synapse: a microservices architecture for heterogeneous-database web applications. In *Proceedings of the Tenth European Conference on Computer Systems*, page 21. ACM, 2015.

[170] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *Computing Colombian Conference (10CCC), 2015 10th*, pages 583–590. IEEE, 2015.

[171] Massimo Villari, Maria Fazio, Schahram Dustdar, Omer Rana, and Rajiv Ranjan. Osmotic computing: A new paradigm for edge/cloud integration. *IEEE Cloud Computing*, 3(6):76–83, 2016.

[172] Jeroen Vuurens, Arjen P de Vries, and Carsten Eickhoff. How much spam can you take? an analysis of crowdsourcing results to increase accuracy. In *Proc. ACM SIGIR Workshop on Crowdsourcing for Information Retrieval (CIR'11)*, pages 21–26, 2011.

[173] Alf Inge Wang, Tommy Bjornsgard, and Kim Saxlund. Peer2me-rapid application framework for mobile peer-to-peer applications. In *Proceedings of the 2007 International Symposium on Collaborative Technologies and Systems (CTS'07)*, pages 379–388. IEEE, 2007.

[174] Lin Wang, Lei Jiao, Ting He, Jun Li, and Max Mühlhäuser. Service entity placement for social virtual reality applications in edge computing. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 468–476. IEEE, 2018.

[175] Shiqiang Wang, Tiffany Tuor, Theodoros Salonidis, Kin K Leung, Christian Makaya, Ting He, and Kevin Chan. Adaptive federated learning in resource constrained edge computing systems. *IEEE Journal on Selected Areas in Communications*, 37(6):1205–1221, 2019.

[176] Shu Wang, Jungwon Min, and Byung K Yi. Location based services for mobiles: Technologies and standards. In *IEEE international conference on communication (ICC)*, volume 19, 2008.

[177] Yun Wang. 24 game solution project on github, 2019. https://github.com/MaigoAkisame/enumerate-expressions.

[178] Andreas Wombacher, Peter Fankhauser, and Erich Neuhold. Transforming bpel into

annotated deterministic finite state automata for service discovery. In *Web Services, 2004. Proceedings. IEEE International Conference on*, pages 316–323. IEEE, 2004.

[179] workflow. Workflow patterns. http://www.workflowpatterns.com/, 2017.

[180] Jie Xu, Lixing Chen, and Shaolei Ren. Online learning for offloading and autoscaling in energy harvesting mobile edge computing. *IEEE Transactions on Cognitive Communications and Networking*, 3(3):361–373, 2017.

[181] Jie Xu, Lixing Chen, and Pan Zhou. Joint service caching and task offloading for mobile edge computing in dense networks. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 207–215. IEEE, 2018.

[182] Zhi-Wei Xu. Cloud-sea computing systems: Towards thousand-fold improvement in performance per watt for the coming zettabyte era. *Journal of Computer Science and Technology*, 29(2):177–181, 2014.

[183] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D Stott Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1029–1040. ACM, 2007.

[184] Stephen S Yau and Junwei Liu. Service functionality indexing and matching for service-based systems. In *Services Computing, 2008. SCC'08. IEEE International Conference on*, volume 1, pages 461–468. IEEE, 2008.

[185] I-Ling Yen, Farokh Bastani, Nidhiben Solanki, and Yongtao Huang. Trustworthy computing in the dynamic iot cloud. In *2018 IEEE International Conference on Information Reuse and Integration (IRI)*, pages 411–418. IEEE, 2018.

[186] Shanhe Yi, Zijiang Hao, Zhengrui Qin, and Qun Li. Fog computing: Platform and applications. In *Hot Topics in Web Systems and Technologies (HotWeb), 2015 Third IEEE Workshop on*, pages 73–78. IEEE, 2015.

[187] Shanhe Yi, Cheng Li, and Qun Li. A survey of fog computing: concepts, applications and issues. In *Proceedings of the 2015 Workshop on Mobile Big Data*, pages 37–42. ACM, 2015.

[188] Shanhe Yi, Zijiang Hao, Qingyang Zhang, Quan Zhang, Weisong Shi, and Qun Li. Lavea: Latency-aware video analytics on edge computing platform. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, pages 1–13, 2017.

[189] Chit Wutyee Zaw, Nway Nway Ei, Han Yeo Reum Im, Yan Kyaw Tun, and Choong Seon Hong. Cost and latency tradeoff in mobile edge computing: A distributed game approach. In *2019 IEEE International Conference on Big Data and Smart Computing (BigComp)*, pages 1–7. IEEE, 2019.

[190] Daniel Yue Zhang, Tahmid Rashid, Xukun Li, Nathan Vance, and Dong Wang. Heteroedge: Taming the heterogeneity of edge computing system in social sensing. In *Proceedings of the International Conference on Internet of Things Design and Implementation*, pages 37–48. ACM, 2019.

[191] Qingyang Zhang, Yifan Wang, Xingzhou Zhang, Liangkai Liu, Xiaopei Wu, Weisong Shi, and Hong Zhong. Openvdap: An open vehicular data analytics platform for cavs. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1310–1320. IEEE, 2018.

[192] Quan Zhang, Xiaohong Zhang, Qingyang Zhang, Weisong Shi, and Hong Zhong. Firework: Big data sharing and processing in collaborative edge environment. In *2016*

Fourth IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb), pages 20–25. IEEE, 2016.

[193] Tianchu Zhao, Sheng Zhou, Xueying Guo, and Zhisheng Niu. Tasks scheduling and resource allocation in heterogeneous cloud for delay-bounded mobile edge computing. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–7. IEEE, 2017.

[194] Chao Zhu, Jin Tao, Giancarlo Pastor, Yu Xiao, Yusheng Ji, Quan Zhou, Yong Li, and Antti Ylä-Jääski. Folo: Latency and quality optimized task allocation in vehicular fog computing. *IEEE Internet of Things Journal*, 2018.